Clube Delphi

Edição 168 :: R$ 14,90

DEVMEDIA

ISSN 1517990-7

Testes unitários com DUnit
Aprenda a realizar testes unitários no Delphi

ListView e ListBox para aplicações mobile
Entenda as principais diferenças e aplicações entre os componentes

Serviços RESTful com DataSnap
Construa uma Web API REST no Delphi

MACHINE LEARNING NO DELPHI

Desenvolva aplicações inteligentes com Intelligence Lab

MVP

R$ 1.000.000,00
INVESTIDOS EM CONTEÚDO NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO NA SUA CARREIRA...

E MOSTRE AO MERCADO QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS
R$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

DEVMEDIA

Clube Delphi

168ª Edição 2016 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Joel Rodrigues (joelrlneto@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Consultora Técnica

Daniella Costa (daniella.devmedia@gmail.com)

Capa e Diagramação

Romulo Araujo

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Joel Rodrigues - Editor da Revista
joelrlneto@gmail.com

Joel Rodrigues - **Editor Geral**

Editor da ClubeDelphi Magazine

Sumário

04 – ListBox e ListView: Listagem de dados em aplicações mobile
[*Jones Granatyr e Jean Paul Barddal*]

Conteúdo sobre Boas Práticas

14 – Testes Unitários com DUnit
[*Guilherme Tempesta*]

Destaque - Vanguarda

26 – Aprendizagem de máquina com Intelligence Lab
[*Jones Granatyr*]

Artigo no estilo Solução Completa

38 – Construindo serviços RESTful com DataSnap
[*Gutierry Antonio*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

ListBox e ListView: Listagem de dados em aplicações mobile

Entenda as principais diferenças e aplicações entre os componentes

Fique por Dentro

Desde as primeiras versões do Delphi, o principal componente para exibição de dados em grade foi o DBGrid. Porém, com a mudança de paradigma de desktop para mobile, outras soluções são necessárias para realizar a mesma função de listagem. Nesse artigo, serão demonstrados os componentes disponíveis nas últimas versões do Delphi que possuem os recursos para tal função. Veremos aqui como construir telas de lista e detalhes de dados utilizando controles do tipo ListBox e ListView, em conjunto com o MultiView e PrototypeBindSource para gerar dados aleatórios para testes em um aplicativo Android.

O Delphi sempre foi muito eficaz para o desenvolvimento de aplicativos desktop. Nesse tipo de software, a preocupação dos desenvolvedores com o redimensionamento dos elementos, e a interface gráfica num geral, é menor quando comparada com dispositivos móveis. Tal fato se dá principalmente pela maior capacidade de memória, processamento e tamanho de tela dos desktops, quando comparados com dispositivos móveis.

O principal componente para listagem de dados utilizado pela maioria dos desenvolvedores Delphi tem sido o DBGrid, que está geralmente vinculado a um DataSource. Um controle desse tipo permite exibir informações no formato de linhas e colunas e está presente em grande parte das aplicações desktop Delphi que lidam com bancos de dados, principalmente em telas de consulta e seleção de dados. No entanto, com o surgimento das novas versões do IDE e também com a mudança dos paradigmas de programação (**BOX 1**), novos componentes e formas de desenvolver as interfaces gráficas foram surgindo.

A primeira mudança nesses paradigmas de programação ocorreu quando houve um grande crescimento e demanda por softwares web, fazendo com que as interfaces e a maneira de programar dos desenvolvedores tivesse de ser adaptada. Por exemplo, para construir uma listagem de registros em uma aplicação que funciona em um navegador web, novas técnicas tiveram que ser levadas em consideração e novos componentes tiveram de ser criados para se obter o mesmo efeito de um DBGrid, por exemplo. Enquanto em aplicações desktop normalmente o usuário pode visualizar todos os registros de uma lista apenas rolando as linhas do grid para baixo e para cima, em ambientes web esse scroll nem sempre é possível ou adequado. Para permitir a navegação entre os registros nesse caso, recursos de paginação são aplicados, dividindo os registros em conjuntos menores.

A mudança seguinte na forma de programar se deu com o aumento da demanda por sistemas para smartphones e tablets, pois a maneira de construir uma aplicação web, na maioria das vezes, não é adequada para um aplicativo móvel.

BOX 1. Paradigmas de desenvolvimento

Um paradigma define a forma como a estrutura e desenvolvimento de um programa é visualizada e considerada pelo desenvolvedor. No contexto desse artigo, os paradigmas considerados dizem respeito aos diferentes tipos de aplicações desenvolvidos ao longo dos anos, que antes se resumiam a softwares desktop (de console ou de janelas/formulários) e atualmente se estende à web e aos variados dispositivos móveis. Para cada demanda, alguns padrões de desenvolvimento e interface são utilizados para oferecer a melhor experiência de uso, e podem variar bastante de acordo com a plataforma-alvo da aplicação.

Por exemplo, o formato de listagem de registros em sistemas web, utilizando tabelas em HTML geradas no servidor, deve ser substituído por novas técnicas, como páginas HTML com design responsivo ou aplicativos nativos.

Dispositivos móveis apresentam diversas limitações, às quais os programadores devem estar atentos durante o desenvolvimento de aplicativos. Por exemplo, nesse tipo de ambiente normalmente se tem memória limitada e telas pequenas, o que torna inviável utilizar uma grade de dados com muitos campos, como é feito em softwares desktop ou web. A geração desse tipo de tabela seria problemática pela dificuldade de visualização na tela, mas também como pelo demasiado uso de memória. Além disso, as telas para cadastro também devem possuir poucos campos, e quando apresentarem muitos dados, sua divisão é necessária. Uma abordagem para divisão seria a quebra de uma tela em várias guias em um TabControl, por exemplo, como veremos mais adiante na **Figura 2**.

Outra importante limitação que os desenvolvedores devem ter em mente é que, na maioria das vezes, celulares e tablets não possuem um teclado como um computador comum, portanto, dependendo das dimensões do aparelho, digitar grandes textos se torna uma tarefa custosa e importuna.

Dentro desse contexto, os controles de listagem em aplicações móveis são de suma importância, pois elas desempenham um papel fundamental para a visualização dos dados. Alguns exemplos são listas de contatos, configurações, playlists, bookmarks, álbum de fotos, notícias, mensagens, dentre outros. Controles desse tipo possuem várias funcionalidades, como apresentar detalhes do texto, realizar agrupamentos, adicionar imagens, pesquisas automáticas e até mesmo a incorporação de outros componentes, tais como botões, checkboxes, dentre outros.

No Delphi, o FireMonkey oferece dois componentes principais para o gerenciamento de listas, que são o ListBox e o ListView, cujas principais diferenças serão apresentadas nas próximas seções desse artigo, bem como em que situações cada um deve ser utilizado.

ListBox e ListView

O ListBox é o componente comumente utilizado para janelas de configurações, pois permite a definição de listas simples e com itens fixos e curtos. Por exemplo, quando a janela de configurações de um aparelho celular é acessada, as opções são fixas e dificilmente mudam com o tempo. Outro exemplo são as telas de configuração de aplicativos, como do WhatsApp, que possui as seguintes opções: Ajuda, Perfil, Conta, Conversas e chamadas, Notificações e Contatos. Essas opções são fixas no aplicativo e não têm a tendência de mudar no decorrer do tempo. Por esse motivo, o ListBox permite a personalização individual de cada opção, ou seja, cada um dos itens presentes na lista pode ter suas propriedades totalmente diferentes de acordo com o contexto da aplicação. Um exemplo é a incorporação de outros componentes dentro de cada um dos elementos da lista. Em resumo, o ListBox é um componente com scroll mínimo que deve ser utilizado quando a quantidade de itens da lista é conhecida a priori e não dinâmica, sendo definidos em "design time" pelo desenvolvedor.

Por outro lado, o ListView é um componente que apresenta listas maiores e possivelmente dinâmicas, geralmente com dados provenientes de uma base de dados, semelhante a um DBGrid. Com esse componente é possível listar uma quantidade indefinida de registros e as configurações para cada uma das suas seções é homogênea, apresentando poucas opções para personalização de cada registro. Para listar os dados, ele permite a vinculação com DataSets por meio do recurso de Live Bindings e em geral essas informações são apresentadas em três seções: cabeçalho, detalhe e rodapé.

Baseado nesses conceitos, o objetivo do presente artigo é apresentar como construir aplicativos no Delphi 10 Seattle que utilize esses dois componentes, mostrando como realizar as configurações básicas necessárias para seu funcionamento. Primeiramente, será mostrado como construir uma janela de configurações utilizando o ListBox, apresentando seus principais recursos. Posteriormente, será apresentada a construção de um exemplo com o ListView por meio da geração de dados aleatórios com o componente PrototypeBindSource, o qual permite que vários registros sejam gerados automaticamente sem a necessidade de possuir uma conexão com um banco de dados ou um ClientDataSet. Será também mostrado como trabalhar com o ListView ligado a guias de detalhes em um TabControl, bem como a utilização do componente MultiView integrado com o ListView. Ambos os exemplos serão construídos com o mínimo de codificação possível, aproveitando assim todo o potencial RAD (*Rapid Application Development*) do Delphi.

Simulando uma janela de configurações com o ListBox

A primeira etapa para a construção dos exemplos é criar uma nova aplicação multi-dispositivo no Delphi 10 Seattle (*File>New>Multi-Device Application*) utilizando o layout em branco (*Blank Application*). Quando o projeto é gerado, automaticamente um novo formulário é criado e fica disponível (*frmListBox*), no qual deve ser adicionado um componente TListBox (*lsbConfiguracoes*) com a propriedade *Align* setada para *Client*, para que o mesmo ocupe toda a tela do formulário. A **Figura 1** apresenta o exemplo que será construído, que tem como objetivo simular a interface gráfica de uma tela de configurações.

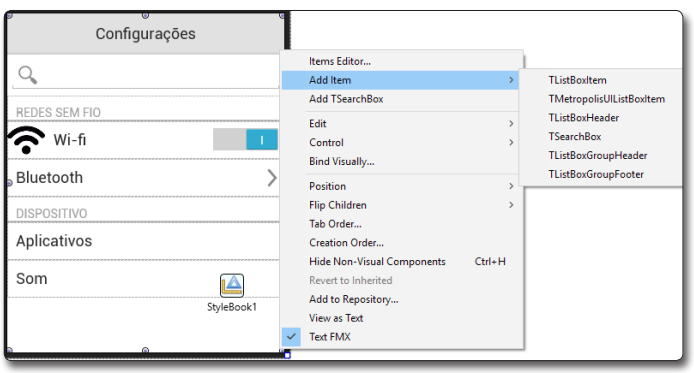


Figura 1. ListBox para configurações

Para que os elementos possam ser adicionados deve-se clicar com o botão direito do mouse no ListBox e acessar a opção *Add Item*, conforme apresentado na **Figura 1**. Os seguintes elementos devem ser adicionados em ordem: TListBoxHeader, TSearchBox, TListBoxGroupHeader, TListBoxItem, TListBoxItem, TListBoxGroupHeader, TListBoxItem e TListBoxItem. Como pode ser observado, o TListBoxHeader apresenta o título da tela e o TSearchBox adiciona uma lupa para fazer pesquisas automáticas nas configurações. Por outro lado, o TListBoxGroupReader (opções *redes sem fio* e *dispositivo*) é utilizado para agrupar os elementos. Por fim, o TListBoxItem (*Wi-fi*, *Bluetooth*, *aplicativos* e *som*) representa as configurações em si, nas quais as funcionalidades serão efetivamente construídas e programadas. É possível notar que também foi adicionado um TStyleBook no formulário, que será posteriormente configurado para melhorar o visual do aplicativo.

Para que o texto "Configurações" seja visível no topo da janela, é necessário adicionar um TLabel dentro do cabeçalho, arrastando-o da paleta de componentes, e configurar sua propriedade *Text*. Já para o item do Wi-fi, no qual é possível observar que há um botão "liga e desliga", basta que seja adicionado um componente TSwitch, o qual é bastante similar às opções usualmente utilizadas em dispositivos móveis. Nesse exemplo nenhuma codificação será realizada, porém, caso fosse necessário realizar alguma programação bastaria acessar e programar o evento *OnClick* para executar as operações. Em aplicativos comerciais, esse tipo de botão é bastante útil em janelas de configuração, onde o usuário é capaz de configurar se determinada função estará ou não disponível no sistema.

Para adicionar a imagem do Wi-fi, deve-se selecionar o ListBoxItem correspondente e acessar a propriedade *ItemData>Bitmap*, selecionando depois o arquivo. Por padrão a imagem não é exibida, pois é necessário configurar a forma de exibição do ListBox por meio da propriedade *DefaultItemStyles>ItemStyle*. Existem diversas opções que dizem respeito a como o ListBox será exibido. Nesse exemplo, para ter o mesmo efeito da **Figura 1**, deve-se selecionar a opção *listboxitemrightdetail*. Outra particularidade que pode ser observada é o item *Bluetooth*, que possui uma seta ao lado indicando que existem mais opções de configuração que usualmente são abertas em outra guia. Esse recurso é configurado por meio da propriedade *ItemData>Acessory* e outros valores podem ser selecionados, como um balão de informação ou um checkmark. O que normalmente ocorre com esse tipo de item é que quando selecionado é feito o direcionamento para uma outra guia que apresentará mais detalhes sobre a configuração. Nesse exemplo essa função de mudança de guias não será abordada pois é assunto para a próxima seção, que por sua vez abordará o ListView.

O componente StyleBook fornece vários estilos pré-definidos para serem adicionados no aplicativo, melhorando o visual sem a necessidade de ter que definir cada componente individualmente. Um duplo clique nesse componente abre uma janela, na qual após selecionar o estilo desejado e clicar em *Apply and Close*, o mesmo é aplicado em todo o formulário. Nesse caso, para ficar igual à **t1** foi utilizado o estilo *AndroidLight*. Além disso, o formulário deve ter sua propriedade *StyleBook* configurada para apontar para o *StyleBook1*.

Nota

Na instalação padrão, os estilos ficam armazenados no seguinte caminho: C:\Users\Public\Documents\Embarcadero\Studio\17.0\Styles.

ListView com TabControl utilizando dados aleatórios

Nesta seção será mostrada a construção de um exemplo que utiliza os recursos visuais do ListView para ligação com dados utilizando o LiveBindings. Para esse exemplo, o primeiro passo é criar um novo formulário (*frmListView*) no Delphi 10 Seattle (*File>New>Multi-Device Form – Delphi*), bem como a adição de um componente TTabControl (*tabPrincipal*) com a propriedade *Align* marcada para *Client*, para que o mesmo preencha toda a área do formulário. Duas guias devem ser adicionadas dentro desse componente, clicando com o botão da direita e escolhendo a opção *Add TTabItem*. Dentro da primeira guia (*tabListagem*) deve ser adicionado um TListView (*lvwDados*) com a propriedade *Align* marcada para *Client*. Por outro lado, na segunda guia (*tabDetalhes*) deve ser adicionado um TLabel (*lblValor*), um TMemo (*mmObservacoes*), um TImage (*imgFoto*) e um TButton (*btVoltar*). O TMemo deve ter sua propriedade *TextSetting>WordWrap* configurada para *True*, para que assim ocorra quebra de linha no texto. A **Figura 2** apresenta a interface gráfica inicial do aplicativo.

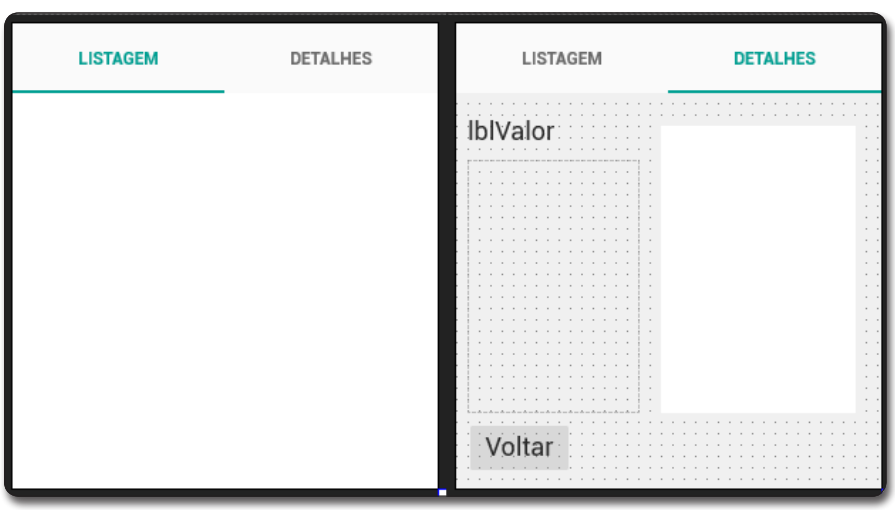


Figura 2. Interface gráfica inicial

As duas guias estão visíveis uma ao lado da outra, porém, o funcionamento do aplicativo seguirá a seguinte lógica: quando a janela for aberta no dispositivo, somente a guia de listagem estará disponível, a qual terá a função de listar todos os registros no componente ListView. Essa guia exibirá alguns dos campos, e quando um

registro for selecionado, a guia listagem será ocultada, dando lugar à guia de detalhes, que mostrará um valor numérico, a foto do usuário e um componente Memo contendo as observações sobre o registro. Quando o botão voltar da segunda guia for acionado, a guia de detalhes será ocultada novamente e dará lugar à tela inicial com a listagem de todos os registros. Essa técnica do gerenciamento das janelas é bastante utilizada em dispositivos móveis, pois como as dimensões do aparelho são limitadas, nem todos os dados poderão ser mostrados ao mesmo tempo. Assim, seria inviável adicionar o Memo com muito texto ou a foto do usuário com alta resolução na primeira guia junto com a listagem dos registros.

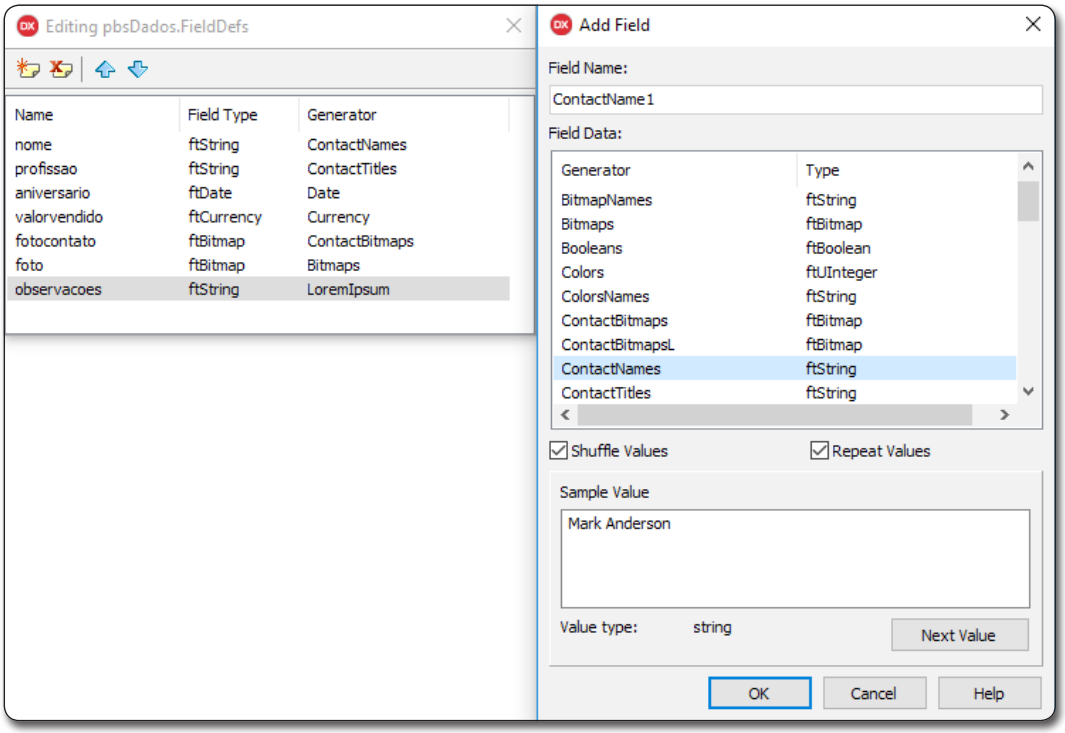


Figura 3. Configuração do PrototypeBindSource

Nota

Nesse artigo optou-se por construir os dois exemplos dentro do mesmo projeto. Para escolher qual dos formulários deverá ser executado, deve-se acessar o menu Project>Options e à esquerda da tela clicar em Forms. Na opção Auto-create forms deve ser escolhido qual deles será executado, arrastando-o para o topo.

Para realizar esses ajustes, é necessário configurar a propriedade *TabPosition* do TTabControl principal (*tabPrincipal*) para o valor *None*. Desta forma, pode-se notar que na parte inferior da janela são adicionados dois pontos (em *"design time"*), um correspondendo a cada uma das guias para permitir a mudança de aba durante o desenvolvimento. Essa configuração é bastante similar à divisão por guias que existe atualmente na maioria dos dispositivos móveis que trabalham com esse conceito e evita programar códigos adicionais para esconder ou exibir as guias. Em seguida, é necessário programar o evento *OnCreate* do formulário, conforme mostrado na **Listagem 1**.

Listagem 1. Inicialização da guia inicial

```
1 procedure TListView1.FormCreate(Sender: TObject);
2 begin
3   tabPrincipal.ActiveTab := tabListagem;
4 end;
```

Na linha 3 dessa listagem é atribuída a primeira guia (*tabListagem*) para o TabControl, a qual será a aba inicial ativa quando o formulário for criado e a janela aberta. É importante salientar que esse procedimento pode ser executado via design mudando essa propriedade manualmente diretamente no componente. Todavia, é recomendado que esse processo seja executado via código, pois, caso sejam feitas alterações na quantidade de guias ou em seu ordenamento, o desenvolvedor muito provavelmente não lembrará de redefinir esse valor corretamente no componente. Em projetos com muitas guias, um problema comum é fazer o *deploy* da aplicação e somente depois do aplicativo já instalado perceber que a guia inicial estava definida de forma errada.

A seguir deve ser adicionado um componente TPrototypeBindSource (*pbsDados*), o qual é responsável por gerar, de forma aleatória, um conjunto de registros para serem utilizados nos testes do aplicativo. Trata-se de uma opção para a criação de protótipos, ou seja, quando deseja-se apresentar para o cliente uma prévia da aplicação sem ter que para isso gastar tempo construindo e preenchendo bases de dados ou tabelas temporárias. Outra vantagem de utilizar esse componente é que o *deploy* do aplicativo para o celular fica mais rápido, não sendo necessário carregar e configurar o arquivo da base de dados completa para o dispositivo.

Um duplo clique no componente *pbsDados* abre a janela para seleção dos campos a serem gerados, conforme pode ser observado na **Figura 3**. Os campos são adicionados clicando no primeiro botão à esquerda da imagem (*Add new*), dando origem à janela apresentada à direita, na qual os *templates* podem ser escolhidos. Pode-se observar que cada campo possui um nome, um tipo de dados e um gerador. De particular interesse nesse exemplo é a propriedade *generator*, que indica o *template* ou o assunto a que o campo está relacionado. Por exemplo, no lado esquerdo da **Figura 3** pode-se observar que foram adicionados sete campos com

geradores diferentes: o gerador *ContactNames* é utilizado para gerar nomes de pessoas, o *ContactTitles* para nomes de profissões, *Date* para datas, *Currency* para valores monetários e *ContactBitmaps* para pequenas imagens que são utilizadas nas listas de contatos, como no WhatsApp, por exemplo. O gerador *Bitmaps* é responsável pela geração de imagens aleatórias, as quais podem ser utilizadas como fotos; e por fim, o *LoremIpsum* provê textos aleatórios mais longos relacionados a descrições, em geral apresentadas em campos do tipo memorando.

Além dos geradores utilizados nesse exemplo, o PrototypeBindSource apresenta mais uma série de opções diversas que podem ser muito úteis para a prototipação de aplicativos mais complexos.

A propriedade *AutoActivate* do *pbsDados* deve ser configurada para *True*. Dessa forma, os dados serão carregados automaticamente quando o aplicativo for iniciado, semelhante à propriedade *Active* de um ClientDataSet, por exemplo. Outra propriedade bastante útil é a *RecordCount*, na qual pode ser configurada a quantidade de registros a serem gerados. Após um campo ter sido adicionado é possível realizar alterações acessando a propriedade *FieldDefs* do PrototypeBindSource, sendo aberta a mesma janela apresentada na **Figura 3**. Nesta janela, o campo *nome* deve ter a sua propriedade *Options>optRepeat* marcada para *False*, para que nomes repetidos não sejam gerados. Todos os outros campos permitirão a inclusão de dados duplicados, por meio da configuração desta propriedade alterada para *True*.

Para fazer a ligação dos dados do protótipo com o ListView será utilizado o recurso de Live Bindings, o qual é mostrado na **Figura 4** à esquerda. Pode-se observar que a profissão está ligada ao *ItemHeader.Text*, enquanto que o nome está ligado ao *Item.Text*. O efeito destas ligações pode ser observado à direita da **Figura 4**, na qual podemos ver os nomes das pessoas agrupados pela profissão, que aparece no cabeçalho. É possível também observar que automaticamente é adicionado o componente *BindingsList1* no formulário, o qual armazenada as ligações feitas visualmente.

O próximo passo é adicionar no ListView mais alguns campos e para isso deve-se acessar sua propriedade *ItemAppearance*, a qual permite que o cabeçalho, rodapé e a área de listagem do componente possam ser configuradas. Por exemplo, as propriedades *FooterHeight*, *FooterAppearance*, *HeaderHeight* e *HeaderAppearance* são responsáveis pelas configurações do cabeçalho e rodapé, sendo possível configurar a altura e a aparência. Nesse exemplo é de grande importância a propriedade *ItemAppearance*, que é responsável pela configuração da aparência gráfica dos objetos, conforme demonstrado na **Tabela 1**. Essa tabela apresenta o valor da propriedade juntamente com os objetos que serão visíveis no ListView.

A escolha de alguma dessas propriedades implica em formas diferentes de visualização dos detalhes do ListView. Por padrão, o valor *ListItem* é atribuído para o ListView, o qual apresenta um rótulo e um botão gráfico. Isto pode ser observado na **Figura 4**, na qual abaixo do cabeçalho que apresenta a profissão *Software Engineer*, está presente o rótulo *Marry Harris* e a seta representando o botão gráfico. A cada vez que uma dessas opções é marcada, os campos do ListView tornam-se visíveis (ou ocultos) para realizar as ligações. Nesse exemplo será utilizada a propriedade *ImageListItemBottomDetail*, e a **Figura 5** apresenta o editor de Live Bindings com as novas propriedades e as novas ligações realizadas, bem como o resultado do formulário.

Propriedade	Objetos visíveis
Custom	Utilizado para customizar a aparência de forma diferente das opções padrões
ImageListItem	Uma imagem, um rótulo e um botão gráfico
ImageListItemBottomDetail	Uma imagem, um rótulo, um texto de detalhe e um botão gráfico
ImageListItemBottomDetailRightButton	Uma imagem, um rótulo, um texto de detalhe e um botão textual
ImageListItemRightButton	Uma imagem, um rótulo e um botão textual
ListItem	Um rótulo e um botão gráfico
ListItemRightDetail	Um rótulo, um texto de detalhe e um botão gráfico

Tabela 1. Configuração da aparência do ListView

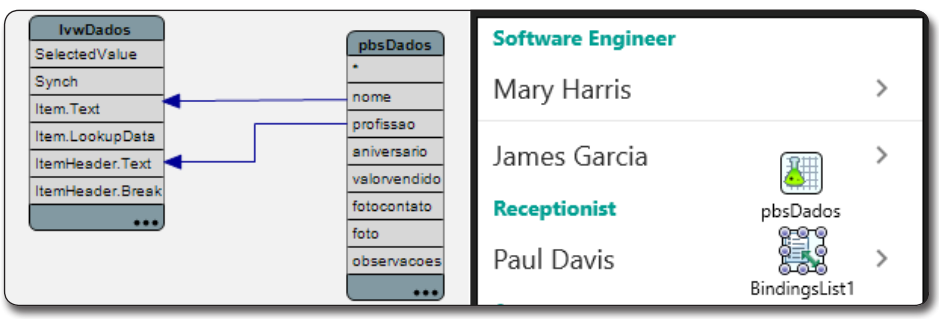


Figura 4. Ligação dos dados ao ListView

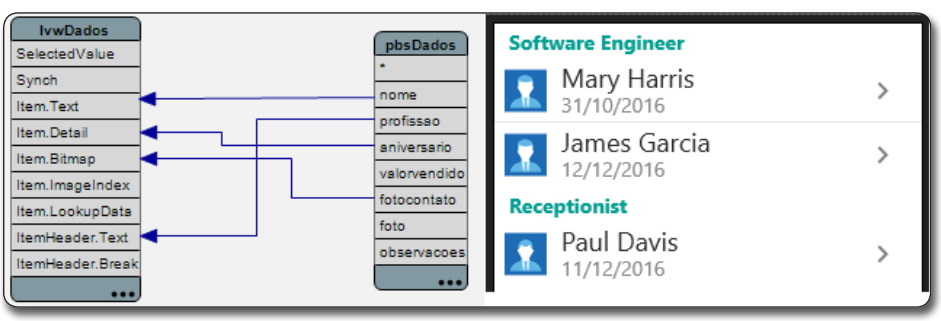


Figura 5. Ligação dos campos nas novas propriedade do ListView

Conforme explanado na **Tabela 1**, o valor *ImageListItemBottomDetail* permite que sejam visíveis um texto de detalhe e uma imagem além dos dados anteriormente vinculados. Desta forma, pode-se observar na **Figura 5** que o campo aniversário foi vinculado ao *Item.Detail* do ListView, enquanto a foto do contato foi vinculada ao *Item.Bitmap*. O resultado pode ser visto nesta mesma figura à direita, na qual a data de nascimento aparece abaixo do nome e a foto ao lado. Nesse contexto, é preciso enfatizar que o PrototypeBindSource gerou as datas de nascimento aleatoriamente, contudo, todas com o ano de 2016.

Nota

Conforme apresentado na **Tabela 1**, a opção Custom permite personalizar a aparência dos itens. Desta forma, caso seja de interesse trocar o posicionamento ou alinhamento dos objetos de forma avançada, essa opção deve ser utilizada para criar estilos personalizados por meio do FireMonkey Style Designer.

As propriedades relativas ao *ItemAppearance* dizem respeito às configurações de posicionamento e tamanho. Por outro lado, as propriedades *ItemAppearanceObjects* são relacionadas às configurações visuais de cada um desses elementos. Por exemplo: *FooterObjects* e *HeaderObjects* disponibilizam diversas opções para formatação do texto do rodapé e do cabeçalho, respectivamente. Como nesse ListView foi vinculado via LiveBindigs somente o cabeçalho, a propriedade *HeaderObjects>Text>TextColor* pode ser modificada para a cor *Slateblue*, por exemplo. Com isso, os cabeçalhos com os títulos das profissões serão todos modificados. De forma similar, o conjunto de propriedades *ItemObjects* possui opções para modificar o botão acessório (*Acessory*), os dados de detalhe (*Detail*), a imagem (*Image*) e o texto (*Text*). Por exemplo, a opção *ItemObjects>Acessory>AcessoryType* permite que o ícone do botão acessório seja modificado para um checkmark ou um ícone de informações (*Detail*), ao invés da seta de "'mais informações" definida como padrão (opção *More*). Caso não seja necessário exibir esses ícones em um ListView, basta configurar a propriedade

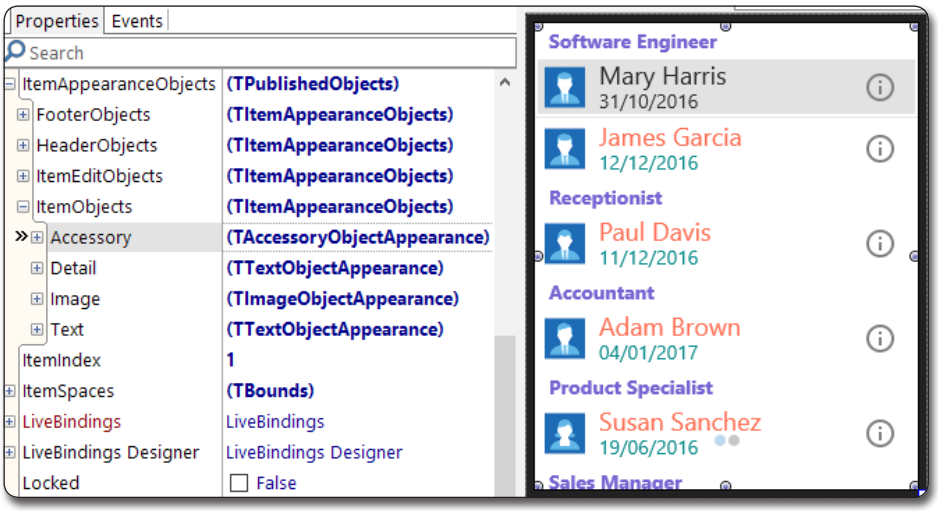


Figura 6. Propriedades ItemAppearanceObjects

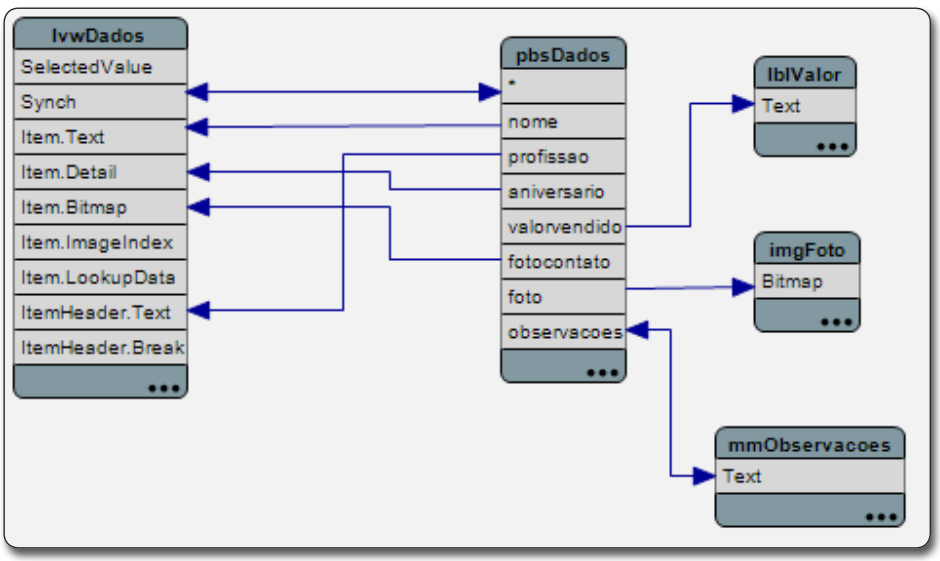


Figura 7. Ligação dos campos da segunda guia

Visible para *False*. De forma similar, o conjunto de propriedades *Detail* realizam as alterações no local que foi reservado para o campo data de nascimento, enquanto que configurações feitas no conjunto de propriedades *Text* serão refletidas no local onde estão listados os nomes das pessoas. Para exemplificar essas alterações, a **Figura 6** apresenta o visual do formulário após elas terem sido realizadas, bem como o Object Inspector mostrando o conjunto dessas configurações.

O próximo passo é a configuração dos campos da segunda guia (*tabDetalhes*), utilizando também o recurso de Live Bindings. A **Figura 7** apresenta as demais vinculações, ou seja: o valor vendido é vinculado ao *lblValor*, a foto ao *imgFoto* e o campo de observações é vinculado ao *mmObservacoes*. É também possível observar que existe um vínculo adicional do asterisco (*) do gerador de dados (*pbsDados*) para a propriedade *Synch* do ListView (*lvwDados*). Essa vinculação é necessária para que o ListView e o PrototypeBindSource estejam sincronizados, ou seja, quando uma mudança de guia for realizada os valores vinculados irão se manter atualizados em ambos. Caso esse vínculo não seja feito, quando a mudança de guia ocorrer, os valores da primeira guia não serão vinculados com a segunda,

sendo esse um erro bastante comum para iniciantes no desenvolvimento de aplicativos móveis no Delphi. Essa característica também é válida para outros componentes de acesso a dados, tal como um ClientDataset ou um TFDQuery.

A próxima e última etapa para que o aplicativo esteja funcional é efetuar as mudanças de guias mencionadas anteriormente, ou seja: quando um registro for selecionado os seus detalhes serão exibidos na segunda guia. Por outro lado, quando o botão *voltar* da segunda guia for acionado, o aplicativo deve retornar para a guia anterior que apresenta a listagem dos registros. Esse processo será feito por meio do componente TActionList (*actAcoes*), que deve ser adicionado no formulário. Um duplo clique nesse componente abre a janela do editor de ações (**Figura 8** à direita), e acessando a opção *New Standard Action* é possível escolher ações padrões específicas para o gerenciamento das guias, não sendo necessárias codificações adicionais, como explanado anteriormente. Nesse exemplo, será utilizada a ação *TChangeTabAction*, pois ela possibilita o controle automático das mudanças de guias em um TabControl. É possível observar que foram definidas as ações *actDetalhes* e *actListagem*. A primeira deve ter sua propriedade *Tab* configurada para *tabDetalhes*, enquanto que a segunda deve ter essa mesma propriedade marcada para *tabListagem*.

Essa configuração é suficiente para que a mudança de guias seja efetuada, bastando agora programar as ações específicas que chamarão esses eventos. A linha 3 da **Listagem 2** apresenta a invocação da ação *actDetalhes*, que levará o controle da aplicação para a segunda guia. Esse código foi definido no evento *OnItemClick* do ListView, ou seja, quando um registro for clicado a mudança de guia será realizada e os detalhes do registro selecionado serão apresentados.

A última configuração a ser realizada para que o aplicativo fique com todas as funções propostas funcionais é atribuir para a propriedade *Action* do botão voltar da segunda guia (*btVoltar*) a ação *actListagem*. Essa configuração irá fazer com que quando acionado, o aplicativo volte para a primeira guia. Ainda um último ajuste opcional é marcar no ListView (*lvwDados*) a propriedade *SearchVisible*, para que um campo de pesquisa seja automaticamente adicionado no topo do ListView. Desta forma, a pesquisa por registros é feita de forma automática sem uma única linha de código. A **Figura 9** apresenta o aplicativo completo sendo executado em um dispositivo Android. É importante salientar que a pesquisa é feita pelo nome da pessoa, ou seja, pelo campo que foi associado ao *Item .Text* no Live Bindings.

Listagem 2. Mudança de guia no TabControl.

```
1 procedure TListView1.lvwDadosItemClick(const Sender: TObject; const AItem: TListViewItem);
2 begin
3   actDetalhes.ExecuteTarget(Self);
4 end;
```

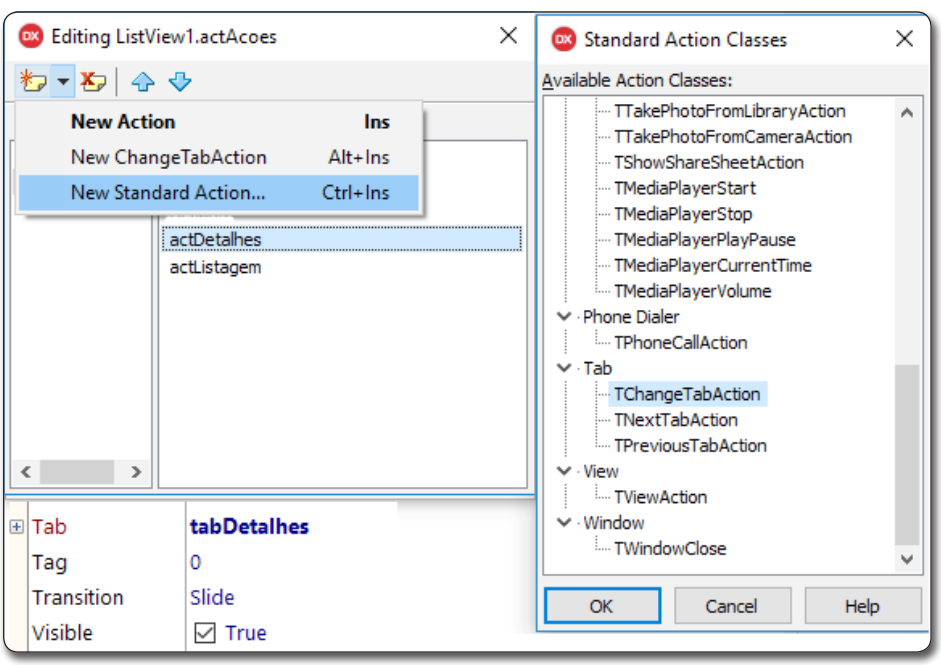


Figura 8. Ações para mudança de guias

Visualização de dados com MultiView e ListView

Nesta seção será mostrada a construção de um exemplo semelhante ao anterior, porém, com a utilização do ListView e sua integração com o componente TMultiView.

O MultiView representa um container no qual outros componentes podem ser adicionados, possibilitando diversas formas de visualização dos dados, principalmente no que tange a janelas do tipo "mestre detalhe" (*master detail*). A estrutura da interface desse componente é composta por um painel mestre e um painel de detalhe. Geralmente, no primeiro são adicionados conjuntos de componentes visuais, enquanto que no segundo painel são tipicamente mostradas informações referentes às propriedades configuradas nos controles do painel mestre. Exemplos típicos da utilização desse componente são janelas nas quais existe um menu de configuração no lado esquerdo (painel mestre) e um painel no centro (painel de detalhe) nas quais as opções correspondentes são apresentadas. Nesse contexto, o painel mestre pode se sobrepor ao painel de detalhe quando acionado; procedimento esse que será mostrado no exemplo a seguir.

O primeiro passo para criar um exemplo utilizando esses componentes é criar um novo formulário (*frmMultiView*) no Delphi 10 Seattle (*File>New>Multi-Device Form – Delphi*) e a adicionar o componente TPrototypeBindSource (*pbsDados*) para a

geração dos valores aleatórios para posterior visualização dos dados. Seguindo os mesmos passos do exemplo anterior, serão adicionados quatro campos: nome (*ContactNames*), profissao (*ContactTitles*), fotoContato (*ContactBitmaps*), observacao (*LoremIpsum*) e foto (*Bitmaps*). O campo *nome* deve ter sua propriedade *Options>optRepeat* marcada para *False* para que não sejam gerados nomes de pessoas repetidos. A **Figura 10** apresenta o formulário completo com todos os componentes visuais adicionados, no qual o componente MultiView (*mtvMestre*) está localizado à esquerda do formulário, e, dentro dele, está presente um ListView (*lsvContatos*) com sua propriedade *Align* configurada para *Client*. No restante da janela encontra-se um componente TLayout (*lytDetalhe*), o qual será o container de todos os outros componentes e tem sua propriedade *Align* marcada para *Client*. No topo da janela encontra-se uma ToolBar (*tlbPrincipal*) e dentro dela um SpeedButton (*spdMaster*) com sua propriedade *Align* configurada para *Client* e o *StyleLookup* setado para *detailstoolbutton*. Essa última propriedade é responsável por mudar o layout do botão, adicionando o ícone de detalhes, sendo que o ícone pode estar visível somente em tempo de execução. Na porção central da **Figura 10** encontram-se os componentes visuais de detalhe, ou seja: TImage (*imgFoto*), dois TLabels (*lblNome* e *lblProfissao*) e um TMemo (*mmObservacao*).

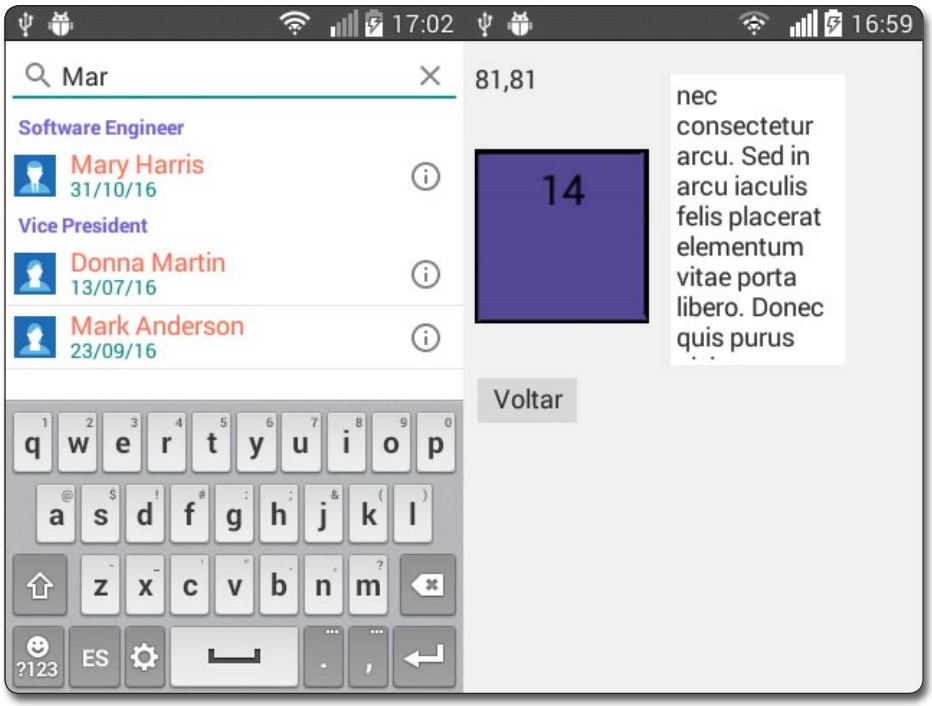


Figura 9. Execução do aplicativo em dispostivo Android

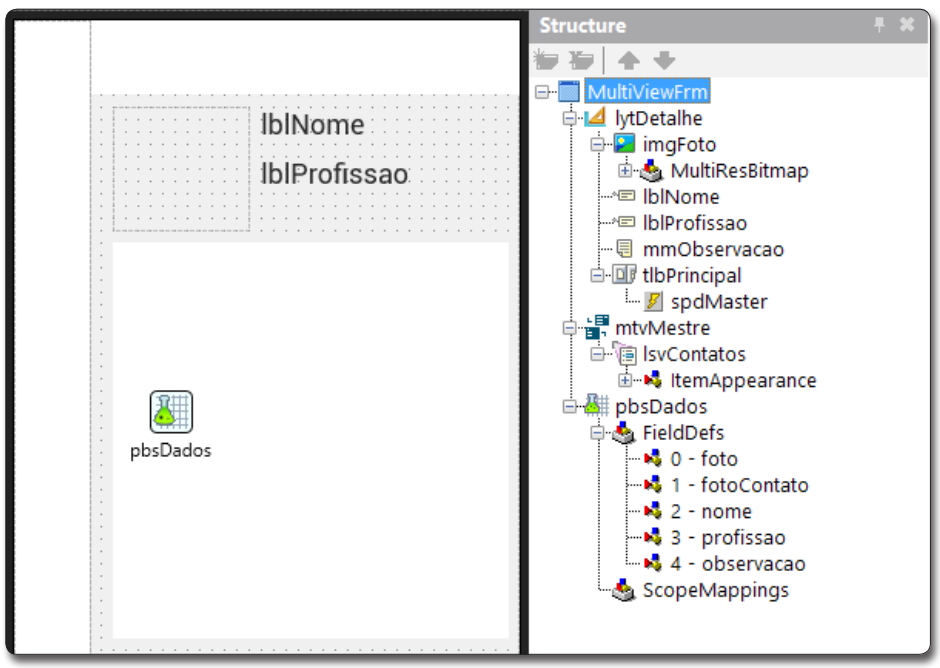


Figura 10. Componentes da janela MultiView e ListView

Nota

O FireMonkey disponibiliza alguns componentes específicos para a criação de layouts com visual complexo, como layouts de grade, de fluxo e com painéis, similarmente aos controles da biblioteca Swing da linguagem Java. Na seção de **Links** encontra-se um material que explana cada um dos tipos de layouts presentes no FireMonkey.

No lado direito dessa figura é possível observar a janela que apresenta a estrutura de todos os componentes, sendo particularmente útil para verificar se todos eles estão em suas posições correspondentes. Para que a janela se torne funcional, é necessário configurar a propriedade *MasterButton* do MultiView (*mtvMestre*) para que a mesma se ligue ao SpeedButton (*spdMaster*). Esse procedimento faz com que quando clicado, as janelas mestre e detalhe sejam sobrepostas uma a outra, conforme descrito anteriormente.

Nesse exemplo, o objetivo inicial do ListView é apresentar somente a foto do contato. Desta forma, o usuário poderá selecionar um registro para que assim os outros dados sejam apresentados na página de detalhe. Para que isso seja possível, a propriedade *ItemAppearance* deve ser alterada para *ImageListItem*, pois conforme observado na **Tabela 1**, essa propriedade permite a adição de uma imagem, um

Listagem 3. Evento ItemClick do ListView

```
1 procedure TMultiViewFrm.lsvContatosItemClick(const Sender: TObject; const AItem: TListViewItem);
2 begin
3   mtvMestre.HideMaster;
4 end;
```

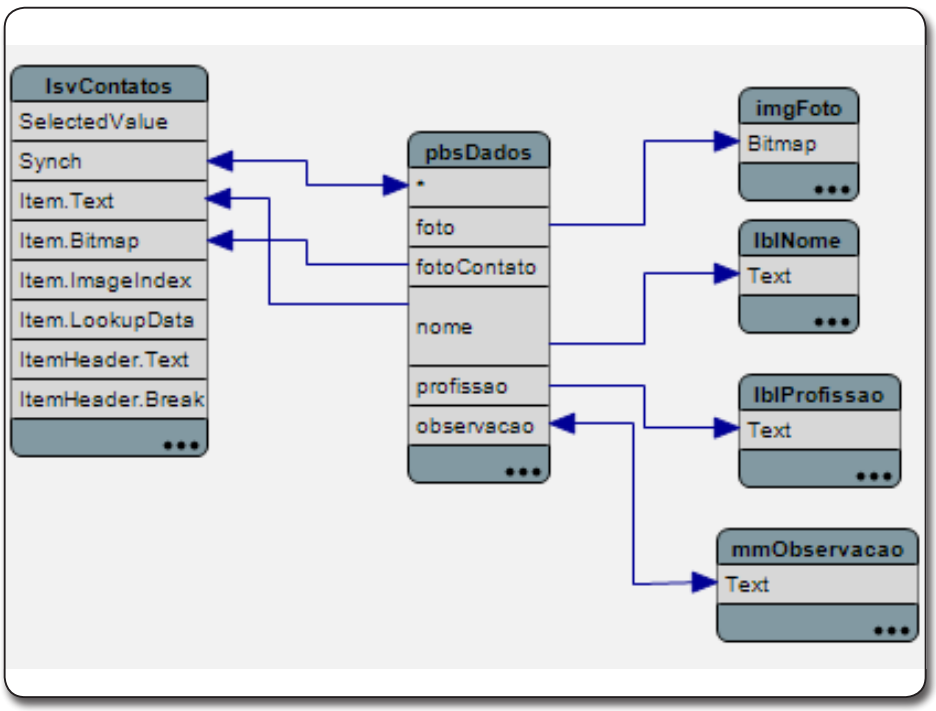


Figura 11. LiveBindings da janela mestre e detalhe

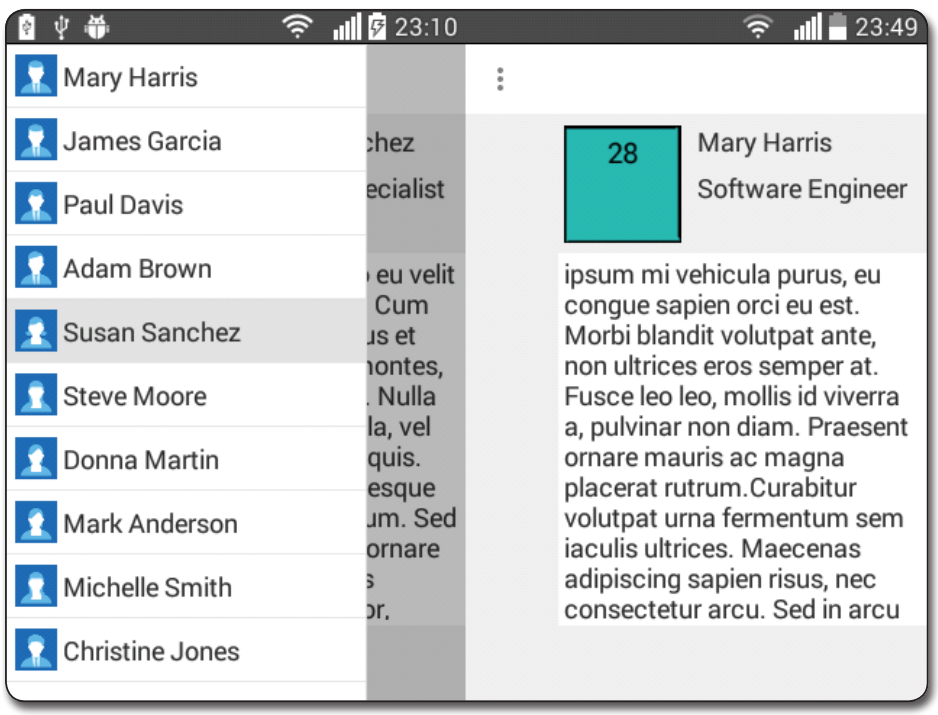


Figura 12. Execução do aplicativo com MultiView

rótulo e um botão gráfico. Além disso, a propriedade *ItemAppearanceObjects>ItemObjects>Acessory>Visible* deve ser marcada para *False*, para que o botão acessório não seja exibido ao lado das fotos.

A **Figura 11** apresenta a vinculação de todos os componentes por LiveBindings.

É possível observar que o asterisco (*) foi vinculado com a propriedade *Synch* para que os componentes visuais fiquem sincronizados ao ListView, enquanto que a foto do contato foi vinculada à imagem e o nome ao texto principal do ListView. De forma similar, os outros campos foram vinculados aos componentes visuais correspondentes na janela de detalhe. Uma última melhoria para a finalização do exemplo é mostrada na **Listagem 3**, na qual foi programado o evento *ItemClick* do ListView. Esse evento é disparado quando um registro do ListView é clicado, e na linha 3 o painel mestre é ocultado.

A **Figura 12** apresenta o aplicativo sendo executado em um dispositivo Android, onde é possível observar o comportamento do componente MultiView. Na imagem à esquerda são apresentados todos os contatos, juntamente com a foto e o nome. Quando um contato é selecionado na página mestre, seus dados automaticamente aparecem na página de detalhe, sendo ocultado o ListView (linha 3 da **Listagem 3**). A característica principal desse exemplo é que quando a página mestre está ativa, ela se sobrepõe à página de detalhe, como pode ser observada a porção cinza da **Figura 12** à esquerda.

Uma propriedade bastante interessante do MultiView é a *Mode*, que indica a maneira como a janela mestre será posicionada. Por padrão, a propriedade *PlatformBehaviour* é selecionada, e ela indica que o comportamento do componente depende do tipo do dispositivo e da orientação da tela. Desta forma, o próprio aplicativo irá se adaptar ao cenário e hardware encontrado. Por outro lado, com o modo *Popover* é necessário clicar no botão definido como *MasterButton* (*spdMaster* nesse exemplo) para que a janela mestre com o ListView se torne visível ou seja ocultada, na forma de um pop-up. Com o modo *Drawer* ativado,

o painel master é inicializado invisível e o usuário deve passar o dedo do canto esquerdo da tela para a direita para que a janela seja exibida. Finalmente, a opção *Panel* apresenta os dois painéis, um ao lado do outro, enquanto que com o modo *NavigationPane* o painel mestre é inicialmente exibido como um pequeno painel encaixado e minimizado no canto da janela.

Nesse artigo, para apresentar as principais configurações do ListView, foi utilizado um componente PrototypeBindSource, porém, um procedimento semelhante poderia ser feito caso fosse necessário trabalhar com um TClientDataSet ou um TFDQuery da biblioteca FireDAC, por exemplo, para lidar com dados reais. Além desses recursos básicos apresentados sobre o ListBox e o ListView para listagem de registros, é possível também fazer personalizações avançadas nesses componentes, bem como a manipulação e seleção de registros e a incorporação de outros componentes. Esse último aspecto é um recurso bastante poderoso, pois permite a customização avançada do componente, permitindo adicionar botões, rótulos, imagens e outras formas avançadas de personalização.

Autor

Jones Granatyr
Doutorando em Informática e Mestre em Ciência da Computação, ambos na área de Inteligência Artificial. Trabalha em projetos de pesquisa relacionados à área de Inteligência Artificial, tais como Sistemas Especialistas, Mineração de Dados, Mineração de Textos e Sistemas Multiagente.

Autor

Jean Paul Barddal
Bacharel em Ciência da Computação e Mestre em Informática pela Pontifícia Universidade Católica do Paraná (PUCPR). Atualmente é doutorando em Informática pela mesma instituição, atuando na área de mineração de dados e aprendizagem de máquina, majoritariamente a partir de fluxos contínuos de dados.

Links e Referências:

Documentação da Embarcadero
http://docwiki.embarcadero.com

Layouts em FireMonkey
http://docwiki.embarcadero.com/RADStudio/Seattle/en/FireMonkey_Layouts_Strategies

DUARTE, William. Delphi para Android e iOS: Desenvolvendo Aplicativos Móveis. Rio de Janeiro: Brasport, 2015.

J. Granatyr, G. Pauli, T. L. Lauxen, J. P. Barddal, F. Spak, F. Taffe. "Construindo Aplicativos Android no Delphi 10 Seattle", Clube Delphi, v. 166, 2016.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Testes unitários com DUnit

Melhore a qualidade do seu sistema com testes

Um sistema que apresenta uma série de erros durante sua execução ou não atende inteiramente as necessidades do usuário possui grandes chances de não obter sucesso. Por isso, os testes de software são indispensáveis para que o produto final apresente o máximo de conformidade possível com os requisitos identificados no início do projeto e atinja um elevado grau de qualidade e confiabilidade.

Erros e falhas (**BOX 1**) podem ser identificados com um menor esforço por parte dos desenvolvedores, ou da equipe responsável pelos testes, ao fazer a escolha por uma estratégia adequada para a execução dos testes de software. Essa deve ser capaz de projetar quando, como e quais testes serão realizados, em cada fase ou etapa do desenvolvimento do software.

BOX 1. Erros e falhas

Apesar de serem muitas vezes utilizados como sinônimos, esses termos podem ser diferenciados, simplificadamente, da seguinte forma: erro é um problema interno no software e se deve normalmente a falhas humanas (como código incorreto) e resultados inesperados (como valores nulos); falha é o comportamento inadequado do sistema, fugindo do que era esperado para ele, e se dá em decorrência de um erro. Ou seja, enquanto erro é um problema interno do software, a falha é o reflexo externo deste.

A criação e a realização dos testes paralelamente com o desenvolvimento do software contribuem para que os erros sejam encontrados o mais breve possível. Isso faz com que não seja necessário esperar que a construção de todo o código-fonte esteja concluída para que se iniciem os testes no software, o que tornaria muito mais complicada a depuração dos erros cometidos na fase de desenvolvimento.

Portanto, é imprescindível que a própria equipe de desenvolvimento tenha consciência da importância de realizar os testes de software simultaneamente com a sua criação, através da execução dos testes unitários, com o objetivo de validar individualmente as menores unidades do código-fonte.

Fique por Dentro

Este artigo tem como foco a realização dos testes unitários em sistemas desenvolvidos em Delphi, mostrando a importância da sua utilização de forma paralela com o desenvolvimento, além da facilidade com que estes testes podem ser implementados pelos próprios desenvolvedores através das funcionalidades de uma ferramenta nativa das versões mais recentes do Delphi, o framework DUnit. Serão criados testes funcionais e estruturais, que poderão ser realizados de forma automatizada ou manual, com o objetivo de mostrar porque os testes unitários são essenciais para a criação de um software totalmente confiável, que funcione de forma adequada e com o mínimo de falhas possível.

Esta atividade colabora diretamente para que o sistema se torne totalmente testável, ou seja, apresente todas as características que facilitam a execução dos testes em todas as suas etapas. Além disso, possibilita a detecção do maior número possível de erros com o mínimo de esforço, reduzindo o custo e o tempo empregado na sua correção, contribuindo para que o resultado final seja um produto que não apresente situações que possam afetar a sua utilização e comprometer sua segurança, eficácia e confiabilidade.

Para que se possa aderir a essas práticas, no entanto, é fundamental a compreensão de conceitos básicos sobre testes de software, que serão apresentados a seguir.

O que são os Testes de Software

Os Testes de Software consistem de uma série de atividades que são realizadas com o intuito de detectar o maior número possível de erros cometidos durante o desenvolvimento de um software, com o objetivo de corrigi-los a fim de garantir que o seu comportamento esteja de acordo com os seus requisitos e funcionalidades definidos durante a concepção do projeto.

Através da elaboração de uma estratégia de testes eficaz e coerente os responsáveis pelo seu planejamento podem determinar a maneira adequada com que os testes de software deverão ser conduzidos. O planejamento dos testes deve utilizar-se de métodos e técnicas de testes eficientes para a criação dos casos de

testes apropriados para cada circunstância definida, de forma que os testes possam ser executados com sucesso. Um teste somente será considerado como bem-sucedido quando localizar um erro que ainda não havia sido encontrado, possibilitando assim sua correção.

Nas próximas seções conheceremos alguns conceitos importantes relacionados a testes de software, cujo conhecimento é fundamental para que se possa aproveitar ao máximo dessas práticas durante o desenvolvimento dos projetos.

Etapas do Teste de Software

Existem diferentes formas para identificar cada uma das etapas percorridas no decorrer da atividade de testes de software. A forma mais habitual, que é utilizada pela maioria dos especialistas no assunto, apresenta quatro etapas distintas:

- **Testes unitários (ou testes de unidade):** esta etapa tem como foco as menores unidades ou componentes de um sistema. Sua finalidade é testar estas unidades individualmente e de forma paralela ao desenvolvimento;
- **Testes de integração:** devem ser realizados durante a integração das unidades. Têm como objetivo analisar se estas operam em conjunto de maneira correta, tal como operam de maneira individual;
- **Testes de validação:** são iniciados depois que as unidades estiverem totalmente testadas e integradas. Visam às operações voltadas para o usuário final, a fim de confirmar a concordância com os requisitos do software;
- **Testes de sistema:** são um conjunto de testes que tem como objetivo testar o sistema como um todo, considerando não só o software em si, mas seu comportamento em conjunto com todos os elementos que formam o ambiente de execução do software, como hardware, banco de dados, redes, etc., e avaliar, sobretudo, seu desempenho e segurança, levando em consideração todos estes fatores externos.

Em cada uma dessas etapas, técnicas específicas podem ser utilizadas para garantir a eficiência dos procedimentos.

Técnicas de testes de software

Nos processos de validação do sistema e identificação de problemas, destacam-se algumas técnicas amplamente utilizadas e extremamente eficientes: os testes caixa-preta e os testes caixa-branca.

Nos testes caixa-preta, o responsável pelos testes não precisa ter conhecimento do comportamento interno do software, ou seja, não necessita visualizar os aspectos estruturais do sistema. Neste caso, o foco é encontrar os pontos em que o sistema está em discordância com suas especificações, através da criação de casos de testes capazes de avaliar o maior número possível de condições de entrada para buscar dados de saída que não estão de acordo com o esperado. Os testes caixa-preta podem ser executados em qualquer etapa dos testes de software.

Já nos testes caixa-branca examina-se a estrutura interna do software, isto é, é necessário analisar diretamente o código-fonte, devendo testar os caminhos percorridos no software ao menos uma vez, através da execução de condições lógicas e ciclos em seus limites operacionais, buscando erros em suas estruturas de dados. Portanto, a técnica de testes caixa-branca é a mais apropriada para ser utilizada durante a fase de testes unitários.

Essas não são, porém, todas as técnicas (ou categorias) existentes. Algumas metodologias são conhecidas e podem inclusive ser adaptadas a cada projeto para melhorar sua execução e adequá-las ao fluxo de trabalho da equipe.

Outras metodologias utilizadas nos Testes de Software

O *Test-Driven Development* (TDD), também conhecido como Desenvolvimento Dirigido por Testes, é uma metodologia que também pode ser empregada durante os testes unitários, cujos testes são criados antes de fazer a implementação do código-fonte do software. Seu objetivo é, através de pequenos ciclos, testar cada requisito do sistema criando um teste simples, rápido e prático.

O ciclo do TDD consiste em alguns passos:

- Desenvolver o novo teste a partir de um requisito do sistema;
- Realizar o teste, que inicialmente deverá retornar uma falha, por ainda não existir implementação;
- Implementar a nova funcionalidade a partir do erro identificado, para que o teste passe;
- Refazer o mesmo teste, sendo que desta vez ele não retornará o erro;
- Fazer a refatoração do código (**BOX 2**);
- Seguir para o próximo teste, sucessivamente.

BOX 2. Refatoração do código-fonte

A refatoração é uma prática utilizada com a finalidade de alterar o código-fonte, aprimorando a sua estrutura interna, para que ele se torne mais organizado, legível e compreensível, facilitando futuras manutenções.

Ao fazer a refatoração, devem ser eliminados quaisquer tipos de duplicidade ou redundância contida no código, evitando que uma mesma funcionalidade seja implementada em vários locais diferentes. Esse tipo de repetição pode fazer com que a manutenção do sistema seja mais trabalhosa e esteja sujeita a falhas, correndo o risco de surgir alguma inconsistência.

Outras vantagens de utilizar a refatoração são a possibilidade de evitar que o código se deteriore depois que alterações são realizadas, a diminuição do tempo gasto para fazer as modificações e ainda a contribuição para que, através de um código organizado e bem estruturado, o desenvolvedor tenha mais facilidade para detectar erros no software.

A utilização do TDD, assim como qualquer outro processo, tem seus pontos positivos e negativos a serem considerados. Alguns pontos positivos do TDD são a capacidade de auxiliar o programador a criar um código mais seguro, com maior qualidade e que atende a cada requisito do sistema. Entretanto, também possui seus pontos negativos, dentre os quais pode-se destacar a mudança de cultura, por ser um processo inverso ao tradicional, podendo causar alguma dificuldade para assimilar as peculiaridades desta metodologia, e o tempo de aprendizado que ele pode requerer. Logo, cabe ao desenvolvedor ponderar a viabilidade de empregar o TDD na estratégia de testes.

Testes unitários em softwares Orientados a Objeto

Os testes de software voltados a aplicações orientadas a objeto possuem algumas particularidades, embora seu objetivo seja o mesmo de quando aplicados em softwares que não seguem esse paradigma.

Especificamente falando sobre os testes unitários, que é o foco deste artigo, o ponto de vista sobre o significado de unidade é alterado, pelo fato de que no paradigma de orientação a objeto existem conceitos a serem considerados, como o de objetos, classes, encapsulamento, herança, polimorfismo.

Neste caso, a menor unidade testável de um software orientado a objeto não seria um módulo individual, mas uma classe encapsulada. Assim, quando uma superclasse possui um método que é herdado por várias subclasses, é preciso testar este método no contexto de cada subclasse existente.

Portanto, o teste convencional seguido ao "pé-da-letra" não tem eficácia se utilizado no paradigma de orientação a objeto, pois o foco não poderá continuar sendo apenas uma operação algorítmica isolada de um módulo específico do código-fonte e seus dados de entrada e saída, mas deverão ser consideradas todas as operações encapsuladas pela classe.

Depuração

A depuração pode ser definida como o processo ordenado, utilizado para corrigir um erro detectado através de um determinado teste, rastreando a sua causa. Ou seja, a depuração remove o erro encontrado por um teste bem-sucedido.

O processo de depuração tem início com a execução dos casos de testes criados, em seguida os resultados apresentados através dos testes são avaliados, e a depuração deverá buscar as possíveis causas da ocorrência do erro.

Quando a causa é identificada, serão feitas as devidas correções e os testes deverão ser novamente realizados, recomeçando um ciclo. Se a causa não for identificada, deverão ser consideradas causas suspeitas, e serão criados casos de testes adicionais para tentar confirmar tais suspeitas, a fim de corrigir o erro de forma iterativa.

Agora que já temos uma boa noção de como os vários tipos de testes são realizados nas etapas de um projeto de software, podemos passar para a parte prática desse artigo e começar a utilizar testes unitários no Delphi.

Planejando os testes unitários

Neste artigo, os testes serão executados em um protótipo de software, que consiste de uma pequena aplicação capaz de calcular o Índice de Massa Corporal (IMC) de um indivíduo. Esta aplicação tem o objetivo de, a partir de dados de entrada, como o peso e a altura, apresentar o valor do IMC, além da sua classificação baseada no valor resultante do cálculo e na faixa etária do indivíduo.

O software no qual os testes serão aplicados é desenvolvido na plataforma Embarcadero Delphi XE8 e utiliza os conceitos de Programação Orientada a Objeto (POO), contendo algumas classes e diversos métodos que servirão de base para a realização dos testes unitários, como a classe *TAvaliacao* e os seus métodos: *CalculaIMC* e *ClassificacaoIMC*, conforme mostrado na **Listagem 1**.

Listagem 1. Classe TAvaliacao.

```
01 type
02  TAvaliacao = class(TObject)
03  private
04   { private declarations }
05   // fields
06   FIdade: Integer;
08   FPeso: Double;
08   FAltura: Double;
09   // getters
10   function GetIdade: Integer;
11   function GetPeso: Double;
12   function GetAltura: Double;
13   // setters
14   procedure SetIdade(const Value: Integer);
15   procedure SetPeso(const Value: Double);
16   procedure SetAltura(const Value: Double);
18  protected
18   { protected declarations }
19  public
20   { public declarations }
21   // encapsulamento
22   property Idade: Integer read GetIdade write SetIdade;
23   property Peso: Double read GetPeso write SetPeso;
24   property Altura: Double read GetAltura write SetAltura;
25   // metodos
26   function CalculaIMC(pAltura, pPeso: Double):Double;
27   function ClassificacaoIMC(pIMC: Double; pIdade: integer): string;
28  published
29   { published declarations }
30  end;
```

Entre as linhas 6 e 16 estão declarados os atributos privados dessa classe e os métodos (*getters* e *setters*) que intermediarão o acesso a eles. Nas linhas 22 a 24 estão as propriedades da classe que serão expostas para clientes externos, encapsulando os elementos privados. Por fim, nas linhas 26 e 27 estão os métodos de cálculo específicos da classe.

Para que se inicie a criação dos testes, o primeiro passo é definir a estratégia que será aplicada no software, tendo em vista a análise alguns pontos importantes, para que seja escolhida a estratégia mais viável, obtendo de tal modo os resultados esperados.

Estratégia de testes

A estratégia de testes escolhida deverá determinar como os testes serão realizados no software, devendo conter algumas informações essenciais como o escopo dos testes, os objetivos que deverão ser atingidos após a sua realização, as etapas que serão contempladas, os tipos de testes que serão executados e os recursos que serão empregados, conforme listado a seguir:

- **Escopo:** Realizar os testes unitários em um projeto simples, desenvolvido no ambiente Delphi, o qual possui algumas classes que serão utilizadas para a realização dos testes;

• **Objetivos:** Estabelecer testes capazes de encontrar o maior número possível de erros cometidos durante o desenvolvimento e, consequentemente, corrigi-los, tendo como foco testar as menores unidades do software que, neste caso, são suas classes e métodos;
• **Etapas de testes:** Será contemplada a etapa de Testes Unitários;
• **Tipos de testes:** Serão realizados os testes estruturais utilizando a técnica de testes caixa-branca, além de basear-se em alguns conceitos da metodologia *Test-Driven Development*;
• **Recursos empregados:** a realização dos testes unitários será feita através do framework DUnitX, que é uma ferramenta disponível no Embarcadero Delphi XE8.

Casos de testes

Os casos de testes deverão ser criados para que o responsável pela realização dos testes, que nesta situação será o próprio desenvolvedor do software, tenha um roteiro a ser seguido e assim possa basear-se em casos de uso UML para descrever situações que ocorrem durante a execução do sistema e programar os testes que serão realizados.

O exemplo exibido na **Figura 1** se trata de um caso de uso simples que apresenta os métodos existentes na classe *TAvaliacao*, referentes ao Cálculo do IMC e a sua classificação:

No caso de teste apresentado, após a inserção dos dados de entrada (idade, peso e altura) pelo usuário por meio da interface do software, a classe *TAvaliacao* deverá fazer o cálculo do IMC através do método *CalculaIMC*, que necessita que os dados de entrada sejam corretamente inseridos. Também deverá ser feita classificação do IMC calculado, através do método *Classificacao-IMC*, o qual depende que o valor do IMC esteja previamente calculado.

Após a criação dos casos de uso de testes necessários, o próximo passo é desenvolvê-los e executá-los no Delphi.

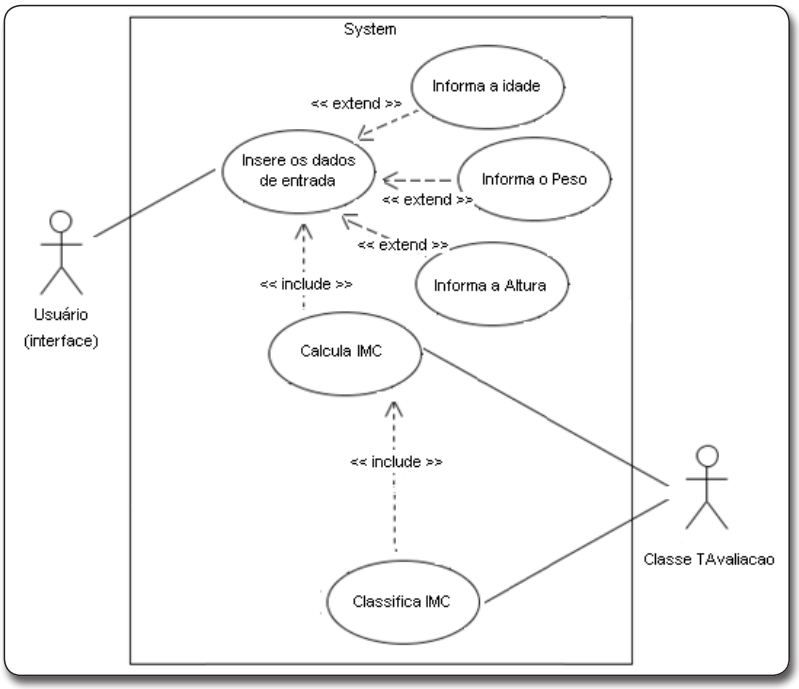


Figura 1. Caso de uso de teste – Calcular e Classificar o IMC

Nome da Função	Descrição
Check	Verifica se uma condição foi cumprida
CheckEquals	Verifica se dois itens são iguais
CheckNotEquals	Verifica se dois itens não são iguais
CheckNotNull	Verifica se um item não é nulo
CheckNull	Verifica se um item é nulo
CheckSame	Verifica se dois itens têm o mesmo valor
EqualsErrorMessage	Verifica se uma mensagem de erro emitida corresponde a uma mensagem de erro especificada
Fail	Verifica se a rotina falhou
FailEquals	Verifica se a falha é igual a uma condição de falha especificada.
FailNotEquals	Verifica se uma condição de falha não é igual à condição de falha especificada
NotEqualsErrorMessage	Verifica se duas mensagens de erro não são as mesmas

Tabela 1. Algumas das funções disponíveis no DUnit.

Testes unitários no Delphi

Apesar de ainda ser relativamente pouco aproveitado por grande parte dos desenvolvedores de software, o Delphi possui uma ferramenta eficiente e de fácil utilização, construída especialmente para a realização dos testes unitários de forma automatizada e integrada, o framework DUnit.

O framework DUnit

O DUnit permite desenvolver e executar testes unitários, de maneira totalmente integrada com as aplicações Delphi, e conta com diversos métodos pré-definidos para testar inúmeras condições distintas, além de ser possível implementar métodos personalizados, conforme a necessidade de cada teste que possa vir a ser realizado.

Nas versões mais recentes do Delphi, o DUnit passou a ser uma ferramenta nativa, ou seja, não necessita de nenhum tipo de instalação, atualização, download ou configuração externa para que o desenvolvedor possa usufruir de todos os seus recursos.

É necessário apenas acessar o menu "*File→New→Other*", e na janela "*New Items*" clicar em *Unit Test*, para que sejam exibidas suas duas opções para iniciar os testes unitários, que são o "*Test Case*" e o "*Test Project*", como mostra a **Figura 2**.

O DUnit apresenta uma série de funções pré-definidas que podem ser utilizadas durante a realização dos testes. Algumas destas funções e suas respectivas definições são mostradas a seguir, na **Tabela 1**.

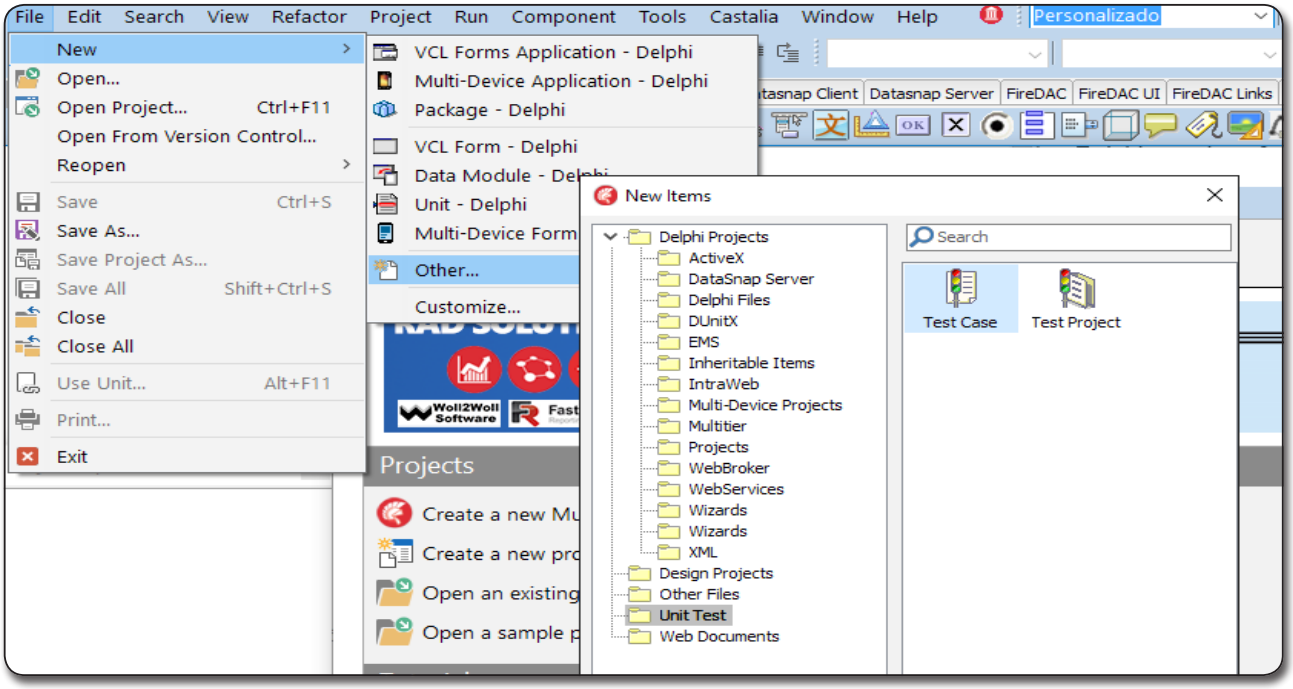


Figura 2. Acessando o menu do DUnit.

Nome da Função	Descrição
Pass	Verifica se uma rotina funciona
Fail	Verifica se uma rotina falha
AreEqual	Verifica se itens são iguais
AreNotEqual	Verifica se itens não são iguais
Contains	Verifica se um item está contido em uma lista
DoesNotContain	Verifica se um item não está contido em uma lista
IsTrue	Verifica se uma condição é verdadeira
IsFalse	Verifica se uma condição é falsa
WillRaise	Verifica se uma exceção será disparada
StartsWith	Verifica se uma sequência inicia com caracteres determinados
IsMatch	Verifica se o item corresponde a um padrão referido

Tabela 2. Algumas funções da classe Assert do DUnitX

A partir da versão do XE8 do Delphi, o DUnit foi descontinuado, ou seja, não serão mais desenvolvidas melhorias ou novas funcionalidades, embora seus recursos ainda estejam disponíveis através do mesmo menu.

Em contrapartida, já existe uma nova versão do framework para testes de soft-ware, o DUnitX.

O DUnitX pode ser considerado uma espécie de evolução do DUnit, pois apresenta várias inovações, dentre elas, o suporte à criação de testes unitários para aplicações multiplataforma (*multi-device*), por ser compatível com os componentes do FireMonkey (**BOX 3**).

Para utilizar o DUnitX no Delphi XE8, basta acessar o menu *"File→New→Other"* e na janela *"New Items"*, entrar na seção *"DUnitX"*, para que sejam exibidas suas duas opções disponíveis: *"DUnitX Project"*, que abre um novo projeto de testes unitários, e *"DUnitX Unit"*, que adiciona uma unit em um projeto já existente. A **Figura 3** mostra o caminho para acessar o DUnitX.

O DUnitX, assim como o DUnit, também disponibiliza várias funções que podem ser utilizadas nos testes, porém estas funções são compatíveis apenas com o DUnitX, não podendo ser utilizadas na versão antiga do framework.

Estas funções podem ser acessadas por meio da classe Assert, e algumas delas são exibidas na **Tabela 2**.

O entendimento dessas funções e de como utilizar o DUnitX é importante para que possamos avançar para a próxima etapa, onde criaremos os testes propriamente ditos no nosso projeto.

BOX 3. FireMonkey

O FireMonkey é um framework presente no Embarcadero RAD Studio, utilizado para o desenvolvimento de aplicações para diversas plataformas como Windows (32 bits e 64 bits), Android, iOS e Mac OS X.

Nota

Atualmente, a própria Embarcadero já aconselha os desenvolvedores a considerarem a migração de projetos de testes antigos, criados utilizando o DUnit, para a nova plataforma do DUnitX, por ser uma ferramenta mais completa e atualizada, suportada nas versões mais recentes do RAD Studio

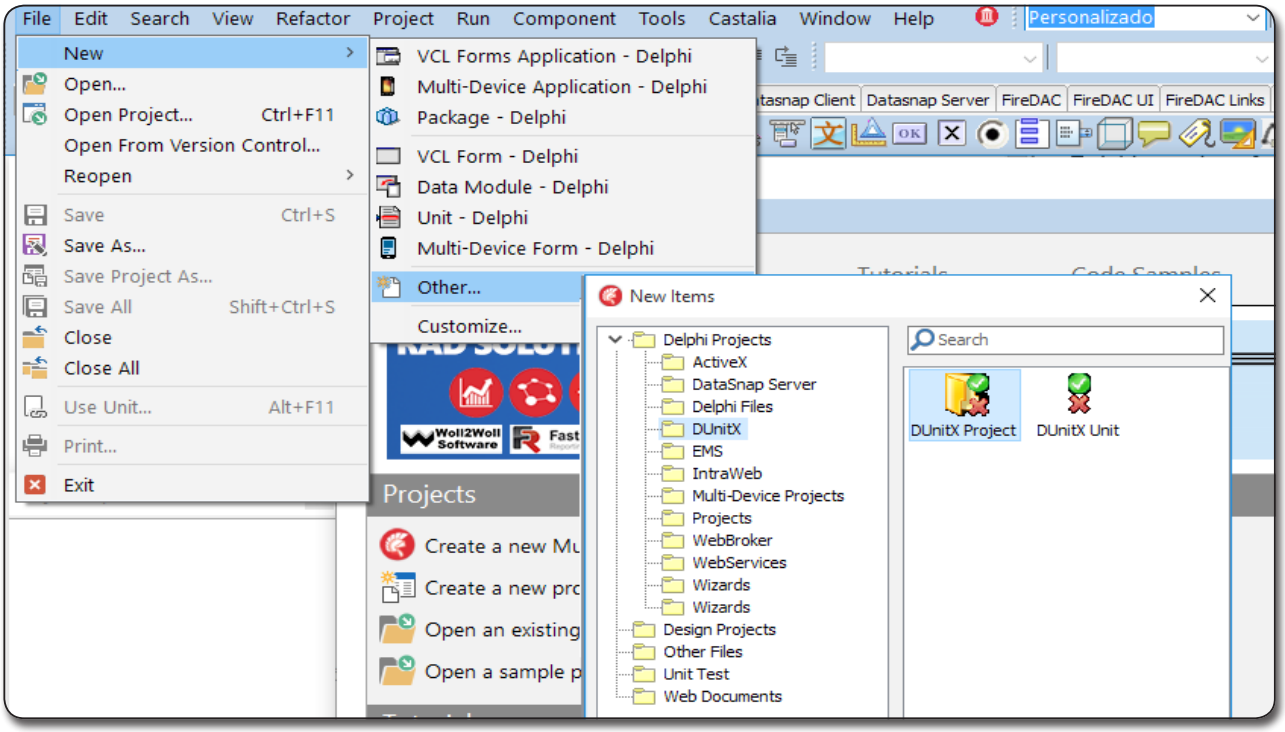


Figura 3. Acessando o DUnitX no Delphi XE8

Executando testes caixa-branca

A partir de agora serão criados testes estruturais utilizando a técnica de testes caixa-branca, que são capazes de avaliar o comportamento interno das estruturas lógicas do software, isto é, testar diretamente o código fonte da aplicação, sem focar somente nos dados de entrada e saída.

Os testes caixa-branca serão criados e executados com o objetivo de encontrar erros no código dos métodos presentes na classe *TAvaliacao*.

Teste do caminho básico

Uma técnica empregada nos testes caixa-branca é o teste do caminho básico, que tem o objetivo de identificar todos os caminhos possíveis a serem percorridos em uma unidade, para que sejam construídos casos de teste para cada um destes caminhos e que eles sejam devidamente testados.

A **Listagem 2** exibe o método *RetornaClassificacaoIMC* da classe *TAvaliacao*, que servirá de base para o teste do caminho básico.

Para submeter esse método ao teste do caminho básico é aconselhável, primeiramente, mapeá-lo de acordo com a sua estrutura de controle, considerando comandos de decisão, repetição e os demais comandos procedurais nele existentes. O mapeamento do código-fonte consiste no agrupamento das linhas, numerando-as de forma que cada conjunto identificado represente um comando que será executado a partir de uma condição distinta. Este agrupamento auxilia o responsável pela criação dos testes a visualizar a estrutura do método de uma forma mais clara, tornando mais fácil a execução dos passos do teste do caminho básico.

Listagem 2. Método utilizado para o teste do caminho básico

```
01 function TAvaliacao.RetornaClassificacaoIMC(pIMC: Double;
02 pIdade: integer): string;
03 begin
04  if (pIdade < 18) then
05   Result := 'Não calculado'
06  else if ((pIdade >= 18) and (pIdade < 60)) then
07  begin
08   if pIMC < 18.5 then
09    Result := 'Abaixo do Peso'
10   else if (pIMC >= 18.5) and (pIMC < 25) then
11    Result := 'Peso Normal'
12   else if (pIMC >= 25) and (pIMC < 30) then
13    Result := 'Sobrepeso'
14   else if (pIMC >= 30) and (pIMC < 35) then
15    Result := 'Obesidade I'
16   else if (pIMC >= 35) and (pIMC < 40) then
17    Result := 'Obesidade II'
18   else Result := 'Obesidade III';
19  end
20  else
21  begin
22   if pIMC < 22 then
23    Result := 'Abaixo do Peso'
24   else if (pIMC >= 22) and (pIMC < 27) then
25    Result := 'Peso Normal'
26   else Result := 'Acima do peso';
27  end;
28 end;
```

Após o agrupamento das linhas do código-fonte, pode ser utilizada a notação do grafo de fluxo, que apesar de não ser obrigatória para realização do teste do caminho básico, é de grande utilidade para que se obtenha um melhor entendimento do fluxo de controle lógico e de sua abordagem, pois disponibiliza uma forma de visualizar graficamente os caminhos existentes.

A notação do grafo de fluxo é representada por um conjunto de nós e arestas, de modo que cada nó do grafo indica um comando procedural e as suas arestas indicam o fluxo de controle, lembrando bastante um fluxograma convencional, porém, muito mais simplificado. Para o método *RetornaAvaliacaoIMC*, por exemplo, seu grafo de fluxo pode ser visto na **Figura 4.**

Com o grafo de fluxo já representado, deve ser efetuado o cálculo da complexidade ciclomática desse método.

O cálculo da complexidade ciclomática é uma métrica de software que, quando utilizada dentro do contexto do teste do caminho básico, tem a função de indicar o número de caminhos independentes existentes na unidade testada e definir a quantidade de testes que precisam ser realizados, para que se possa garantir a execução de todas as condições e comandos pelo menos uma vez.

Para encontrar a complexidade ciclomática do grafo de fluxo exibido anteriormente, na **Figura 4,** foi utilizado o cálculo $V(G) = E - N + 2$, onde $V(G)$ se refere à complexidade ciclomática, E representa à quantidade de arestas e N é a quantidade de nós do grafo, conforme é detalhado a seguir:

V(G) = E – N + 2

V(G) = 28 – 20 + 2

V(G) = 10

O resultado obtido através desse cálculo é igual a 10. Este valor indica que deverão ser projetados dez casos de testes distintos, utilizando os dados que force o teste a percorrer todos os caminhos do grafo de fluxo referente ao método apresentado.

Após realizar o teste do caminho básico, é feito o teste do caminho independente, que identifica cada caminho da estrutura que segue para uma nova condição, determinando um conjunto base com o número de caminhos linearmente independentes.

No grafo exibido na **Figura 4** foram encontrados dez caminhos independentes, que são os seguintes:

- Caminho 01: 1, 2, 20
- Caminho 02: 1, 3, 5, 20
- Caminho 03: 1, 3, 4, 6, 7, 20
- Caminho 04: 1, 3, 4, 6, 8, 9, 20
- Caminho 05: 1, 3, 4, 6, 8, 10, 11, 20
- Caminho 06: 1, 3, 4, 6, 8, 10, 12, 13, 20
- Caminho 07: 1, 3, 4, 6, 8, 10, 12, 14, 20
- Caminho 08: 1, 3, 15, 16, 20
- Caminho 09: 1, 3, 15, 17, 18, 20
- Caminho 10: 1, 3, 15, 17, 19, 20

Nota

Dependendo da complexidade do método, ou seja, se existir um número muito elevado de caminhos lógicos possíveis, pode não ser viável gerar todos os casos de testes necessários. Porém, ainda assim os testes caixa-branca não devem ser descartados, pois existe a possibilidade de determinar a quantidade de caminhos lógicos que serão percorridos, sem prejudicar a viabilidade e a eficácia do teste.

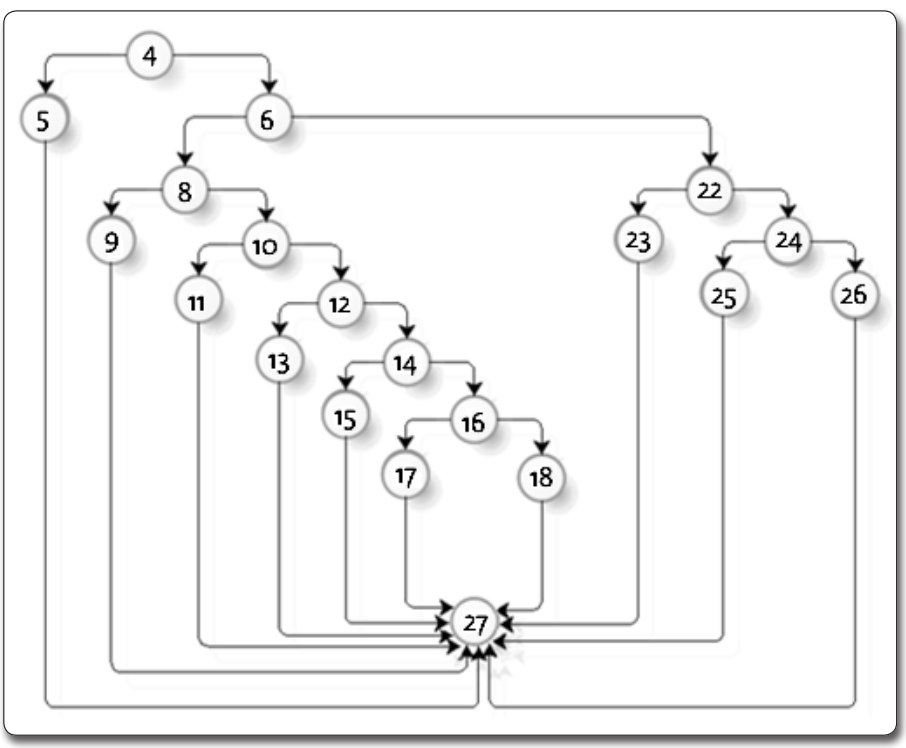


Figura 4. Grafo de fluxo do método RetornaAvaliacaoIMC

Após encontrar os caminhos independentes, deverão ser feitos os casos de teste correspondentes a cada um destes caminhos utilizando o DUnitX.

Primeiramente, deve-se iniciar um novo projeto de teste no Delphi, pelo framework DUnitX, selecionando a opção *DUnitX Project*, conforme já foi mostrado anteriormente e, em seguida, fazer a configuração do projeto de testes, por meio da tela *New DUnitX Project*, exibida na **Figura 5**.

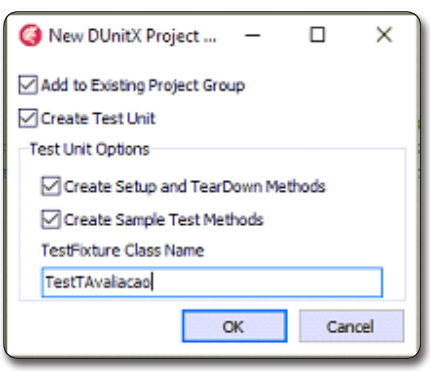


Figura 5. Tela de configuração do DUnitX.

Nesta tela, deverão ser marcados os campos "*Create Setup and TearDown Methods*" e "*Create Sample Test Methods*", além de preencher o campo "*TestFixture Class Name*" com o nome escolhido para a classe responsável pela execução dos testes unitários. Desta maneira, a unit criada pelo DUnitX já irá disponibilizar uma estrutura pré-definida, facilitando a criação dos casos de teste.

A unit criada para executar os testes, nomeada *TestUnit*, contém a classe *TestTAvaliacao*, a qual já possui declarados os métodos *Setup* e *TearDown*, conforme indicado na **Listagem 3**.

Listagem 3. A classe TestTAvaliacao.

```
01 type
02  [TestFixture]
03  TestTAvaliacao = class(TObject)
04  strict private
05   FAvaliacao: TAvaliacao;
06  private
07  public
08   [Setup]
09   procedure Setup;
10   [TearDown]
11   procedure TearDown;
12  end;
```

O método denominado *Setup* é chamado antes da execução dos testes e será responsável por preparar o teste. Neste caso, o *Setup* irá criar o objeto *FAvaliacao*, que é instância da classe *TAvaliacao*, para ser utilizado durante a realização dos testes.

O método *TearDown* é chamado depois de os testes serem executados e é responsável por realizar alguma operação ao finalizar o teste. Neste exemplo, o *TearDown* irá liberar o espaço de memória alocado para o objeto criado durante o método *Setup*.

Sendo assim, o próximo passo será fazer a implementação dos métodos específicos para a realização dos testes para a classe *TAvaliacao* que, a princípio, será o método *TestRetornaClassificacaoIMC*, o qual terá a função de testar todos os dez caminhos independentes que foram determinados para o método *RetornaClassificacaoIMC*.

A **Listagem 4** apresenta como deve ficar a declaração do método de teste na classe *TestTAvaliacao*, sendo que antes da declaração deverão ser indicados os casos de teste fazendo o uso da *tag [TestCase]*, contando com os dados de entrada referentes a cada condição definida nos caminhos independentes identificados no método.

A **Listagem 5**, por sua vez, exibe as linhas de código referentes ao corpo do método *TestRetornaClassificacaoIMC*.

Os parâmetros passados na *tag [TestCase]* são utilizados para executar o método *RetornaClassificacaoIMC*, e seu resultado é armazenado na variável R, do tipo string. Em seguida utiliza-se a função *AreEqual*, disponível através da classe Assert do DUnitX, para comparar os resultados obtidos com o valor passado em cada caso de teste.

Listagem 4. Declaração do método TestRetornaClassificacaoIMC.

```
01 [TestCase('Test01','19,17,Não disponível')]
02 [TestCase('Test02','18,18,Abaixo do Peso')]
03 [TestCase('Test03','22,22,Peso Normal')]
04 [TestCase('Test04','25,35,Sobrepeso')]
05 [TestCase('Test05','30,35,Obesidade I')]
06 [TestCase('Test06','39,59,Obesidade II')]
07 [TestCase('Test07','40,59,Obesidade III')]
08 [TestCase('Test08','21,60,Abaixo do Peso')]
09 [TestCase('Test09','22,60,Peso Normal')]
10 [TestCase('Test10','27,60,Acima do Peso')]
11 procedure TestRetornaClassificacaoIMC2(pIMC: Currency; pIdade: Integer;
    _Result: string)
```

Listagem 5. Corpo do método TestRetornaClassificacaoIMC.

```
01 procedure TestTAvaliacao.TestRetornaClassificacaoIMC2(pIMC: Currency;
    pIdade: Integer; _Result: string);
02 var
03  R: string;
04 begin
05  R := FAvaliacao.RetornaClassificacaoIMC(pIMC,pIdade);
06  Assert.AreEqual(_Result, R);
07 end;
```

Os testes serão executados e, se os valores apresentados corresponderem aos resultados esperados, o DUnitX indicará que o teste passou. Caso contrário, o framework retornará uma mensagem de erro informando que o valor obtido após os testes é diferente do valor esperado.

Ao pressionar a tecla *F9*, o console do DUnitX se iniciará exibindo os detalhes dos testes que foram executados, como a quantidade de testes encontrados, ignorados, que passaram, que falharam, etc. Como o *TestRetornaClassificacaoIMC* é o único teste implementado até o momento, a quantidade de testes que será exibida na tela do console será referente a este método, como pode ser observado na **Figura 6**.

Por meio do resultado apresentado pelo DUnitX, nessa figura, verifica-se que não foi apontado nenhum erro, pois o total de dez testes encontrados, informado na linha *Tests Found*, confere com o total de testes que passaram, informado na linha *Tests Passed*. Desta forma, conclui-se que todas condições apresentadas são verdadeiras.

Posteriormente, poderão ser desenvolvidos os casos de teste para que sejam realizados os testes nos demais métodos da classe *TAvaliacao*, utilizando as técnicas presentes nos testes caixa-branca, da mesma forma como foram realizados no método *TestRetornaClassificacaoIMC*.

Executando testes baseados em TDD

A metodologia TDD também poderá ser utilizada, com o objetivo de criar os testes antes mesmo de desenvolver as funcionalidades que serão testadas, para que os métodos da classe sejam implementados de acordo com os erros que são identificados durante a execução dos testes.

Listagem 6. TestCalculaIMC.

```
01 procedure TestTAvaliacao.TestCalculaIMC;
02 var R: Extended;
03 begin
04   R := FAvaliacao.CalculaIMC(1.77,80,29);
05   Assert.AreEqual(25.54, R);
06 end;
```

Listagem 7. Método CalculaIMC.

```
01 function TAvaliacao.CalculaIMC(pAltura, pPeso: Double;
     pIdade: integer): Double;
02 begin
03   Result := pPeso / (pAltura * pAltura);
04 end;
```

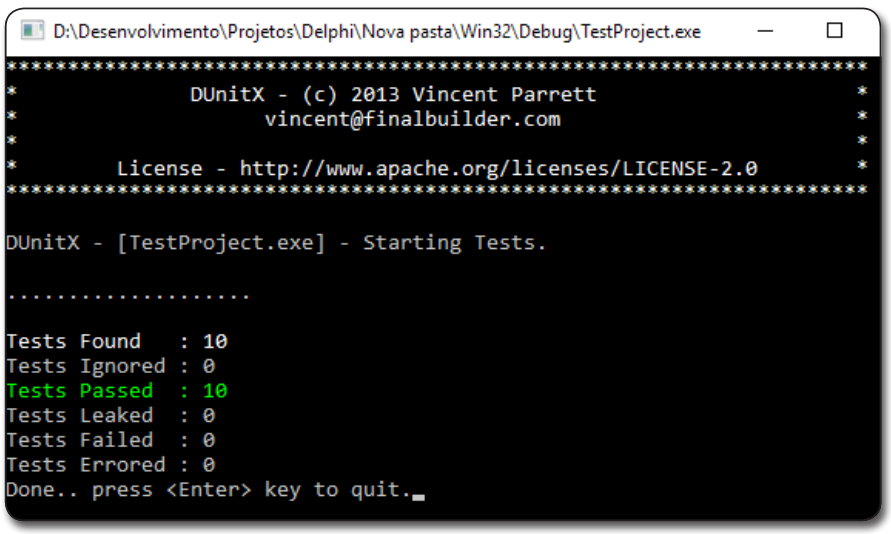


Figura 6. Tela do DUnitX, todos os testes passaram

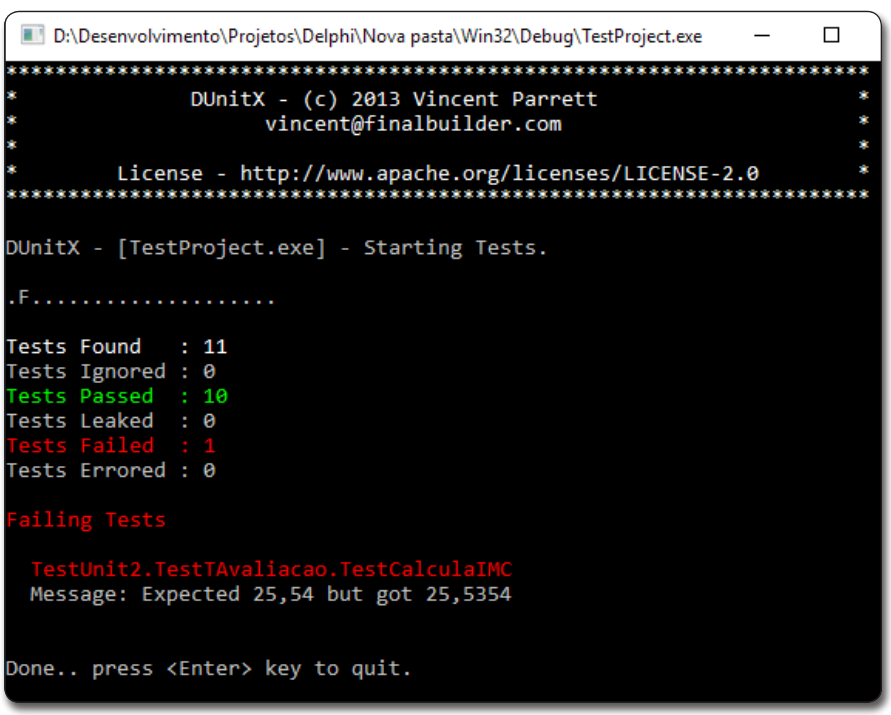


Figura 7. Erro identificado durante o teste TestCalculaIMC

Inicialmente, será criado o teste *TestCalculaIMC* para executar o método *CalculaIMC*, passando valores constantes como dados de entrada, para que o resultado gerado seja comparado com o valor esperado, por meio da função *AreEqual*, como pode ser verificado na **Listagem 6**.

Deverá ser declarada a variável *R*, do tipo *extend*, que receberá o resultado do método *CalculaIMC*. Este método terá como dados de entrada os parâmetros *pPeso* e *pAltura*, do tipo *Double*, e o parâmetro *pIdade* do tipo *integer*; e seu dado de saída será o resultado do cálculo do IMC, que é um dado do tipo *Double*.

Nesta situação, os valores para altura, peso e idade serão: 1.77, 80 e 29, respectivamente, portanto, de acordo com a fórmula para o cálculo do IMC, o valor esperado deverá ser 25.54. A função *AreEqual* necessita de receber os parâmetros *expected*, para o valor esperado, que é o valor informado no teste, e *actual*, que será o valor da variável *R*, ou seja, o resultado do cálculo através do método *CalculaIMC*.

Como o método *CalculaIMC* ainda não foi implementado, então será retornado o primeiro erro, logo na compilação do projeto de testes do DUnitX através do Delphi. Sendo assim, o conceito de TDD já começa a ser utilizado, pois a partir do erro identificado o método será implementado, como pode ser verificado na **Listagem 7**.

Com a implementação do método, os testes serão executados novamente e, desta vez, o DUnitX indicará na tela do console os detalhes e informações dos testes. Como os casos de teste executados pelo método *TestRetornaClassificacaoIMC* anteriormente já foram finalizados, eles irão constar no total de testes que passaram (seção *Tests Passed*). Entretanto, O DUnitX identificou um erro no teste *TestCalculaIMC*, como pode ser observado na **Figura 7**.

De acordo com o DUnitX, este teste falhou (*Tests Failed: 1*) e o erro encontrado após a sua realização mostra que o valor esperado, 25,54 é diferente do valor calculado pelo método *CalculaIMC*, o qual retornou o valor de 25,5354.

Sendo assim, conclui-se que o erro ocorreu porque o valor de retorno do método deveria ser arredondado para duas casas

decimais, o que ainda não foi implementado. O método deve ser alterado, de modo a corrigir o erro apontado pelo teste utilizando a função *SimpleRoundTo*, utilizada para arredondar o resultado, com o objetivo de fazer com que o teste passe da próxima vez que for executado.

O código a seguir mostra como ficou o *Result* do método *Calcula-IMC* após a correção ser efetuada:

```
Result := SimpleRoundTo((pPeso / (pAltura * pAltura)), -2);
```

Após a correção do método e nova execução do teste, o DUnitX não identificou nenhum erro, conforme mostrado na **Figura 8**. Desta forma, o teste *TestCalculaIMC* pode ser considerado como finalizado.

Além de testar a classe *CalculaIMC*, com o objetivo de verificar a igualdade entre os valores do IMC calculado e esperado, utilizando a função *AreEqual*, também surge a necessidade de executar mais um teste referente a outro requisito desta classe. Como a tabela de IMC apresentada nesta aplicação somente suporta os cálculos para adultos e idosos, existe uma exceção que deve ser disparada quando a idade do indivíduo for menor que 18 anos, informando que a faixa etária é inválida para o cálculo.

A função *WillRaise* é utilizada para tratar se uma exceção será disparada conforme o esperado. O teste *TestIMCIdadeMenor18* foi desenvolvido com base nesta função, conforme mostrado na **Listagem 8**.

O teste exibido anteriormente tem o objetivo de verificar se a exceção esperada, *EFaixaEtariaIndisponivel*, será disparada quando o parâmetro *pIdade*, do método *CalculaIMC*, for menor do que 18.

Como a exceção *EFaixaEtariaIndisponivel* não foi implementada no método, o teste não irá passar, pois a exceção não será disparada conforme era esperado. A **Figura 9** exibe o resultado do teste *TestIMCIdade-Menor18*.

A **Listagem 9** mostra o método após a alteração que corrige o erro referente a exceção, identificado no teste anterior.

Para que o teste passe, foi necessário criar uma condição para verificar se o valor da idade é menor que 18.

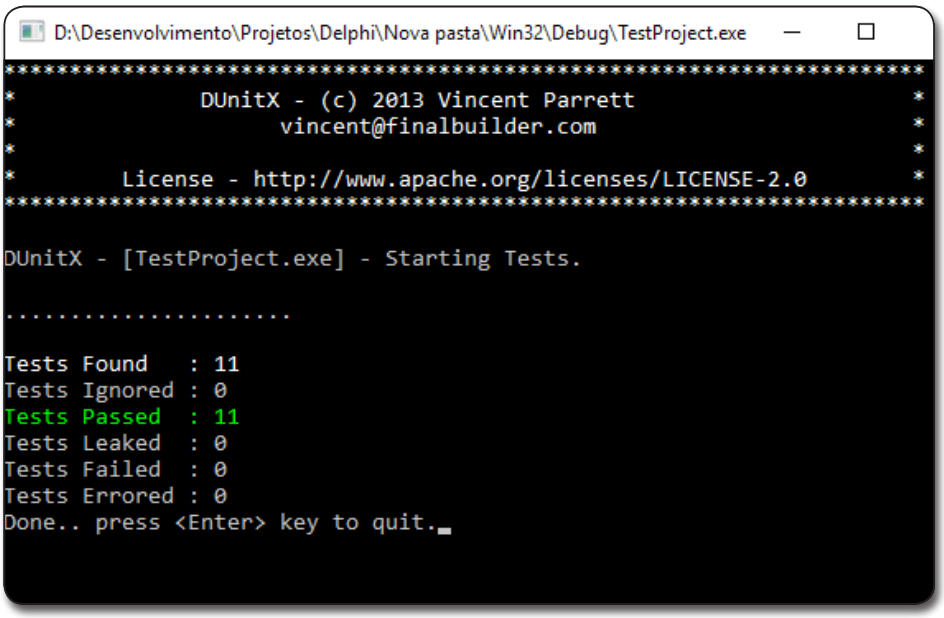


Figura 8. DUnitX indicando que o teste passou

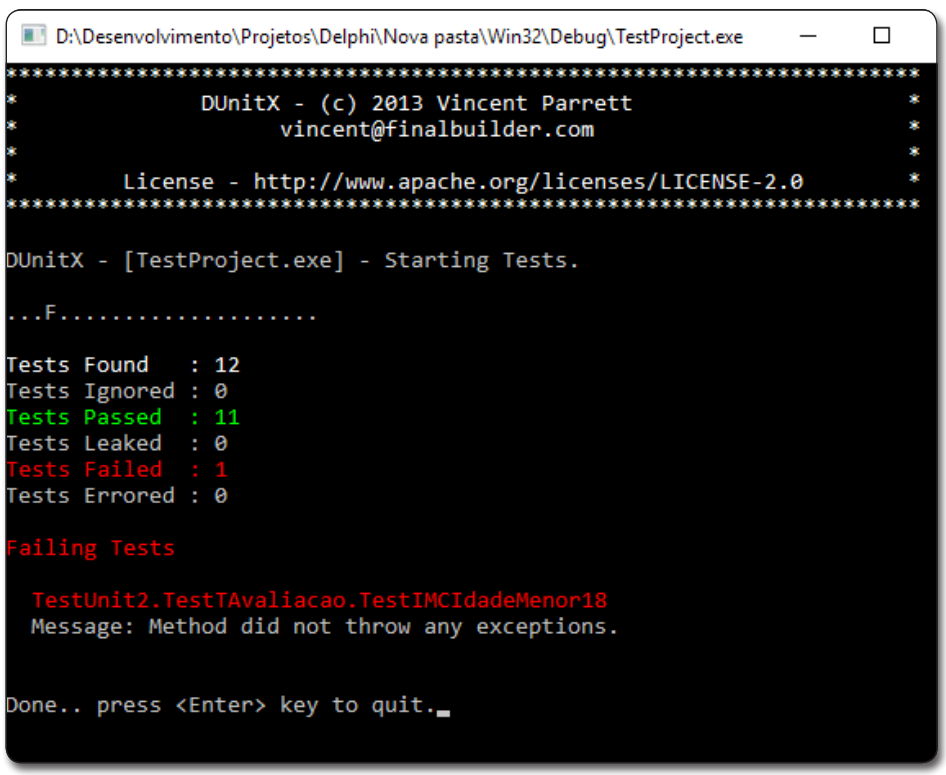


Figura 9. TestCalculaIMC com erro no teste de exceção

Listagem 8. Função WillRaise utilizada no teste.

```
01 procedure TestTAvaliacao.TestIMCIdadeMenor18;
02 begin
03   Assert.WillRaise(procedure
04             begin
05               FAvaliacao.CalculaIMC(1.77,80,17)
06             end, EFaixaEtariaIndisponivel);
07 end;
```

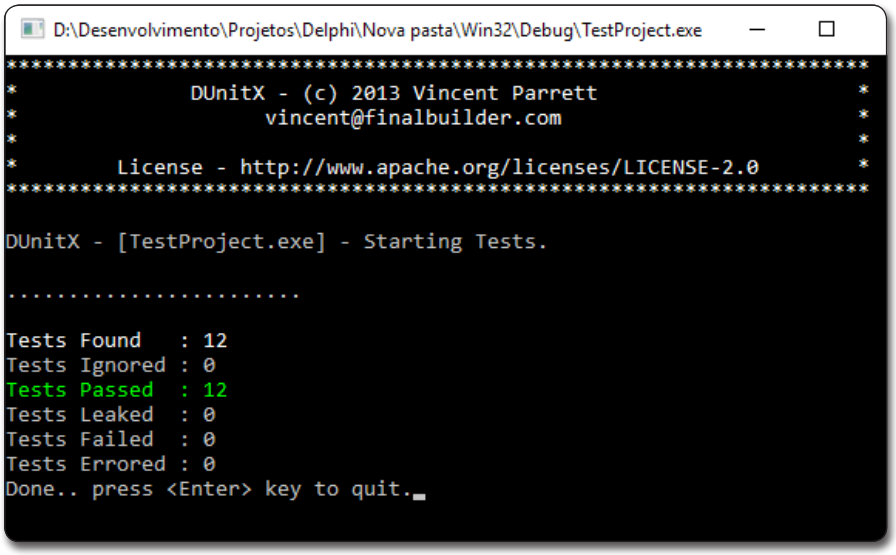


Figura 10. TestCalculaIMC após a correção do erro

Listagem 9. Método CalculaIMC após correção.

```
01 function TAvaliacao.CalculaIMC(pAltura, pPeso: Double;
     pIdade: integer): Double;
02 begin
03   if pIdade < 18 then
04     raise EFaixaEtariaIndisponivel.Create('Faixa etária indisponível.')
05   else
06     Result := SimpleRoundTo((pPeso / (pAltura * pAltura)), -2);
007 end;
```

Se a condição for verdadeira, a exceção *EFaixaEtariaIndisponivel* seja disparada, se for falsa, o cálculo será executado normalmente, sem disparar tal exceção.

Desta forma, ao executar novamente o teste *TestIMCIdadeMenor18*, ele terá que passar. A **Figura 10** mostra a tela do console do DUnitX, confirmando que todos os testes executados até o momento passaram, inclusive o teste da exceção.

Vários testes poderão ser criados considerando os requisitos que deverão ser atendidos pelo software, utilizando a metodologia do desenvolvimento dirigido por testes, da forma com que foram criados e executados os testes no método *CalculaIMC*, da classe *TAvaliacao*.

As funções contidas na classe *Assert* do DUnitX tornam possível a criação de testes mais complexos, fornecendo uma enorme variedade de condições de testes capazes de encontrar grande parte dos erros na fase de testes unitários.

O conjunto de técnicas contempladas pelos testes caixa-branca na realização dos testes unitários, utilizadas da forma correta, faz com que cada unidade do software seja totalmente testada e que ao final do seu desenvolvimento, o seu funcionamento esteja de acordo com o esperado e a maior parte dos erros seja identificado e devidamente corrigido.

Ao finalizar os testes unitários, confirmando que todas as unidades estejam em pleno funcionamento de maneira individual, o próximo passo será fazer a integração das unidades e, consequentemente, testar se as mesmas funcionam de maneira correta também em conjunto. Esta tarefa já pertence a outra fase distinta dos testes de software, que são os testes de integração.

A utilização de todas as metodologias descritas neste artigo para a realização dos testes unitários, combinada com a escolha da melhor estratégia para a execução das fases posteriores da atividade de testes de software, contribui para que a equipe de desenvolvimento obtenha ferramentas e técnicas de apoio que a torne capaz de produzir um software com maior confiabilidade, conformidade e segurança.

Autor

Guilherme Lourenção Tempesta
Bacharel em Ciência da Computação pelo Centro Universitário do Sul de Minas (UNIS-MG) e Especialista em Engenharia de Software, pós-graduado pela Universidade Gama Filho. Desenvolvedor de software em Delphi e banco de dados Firebird.

Links e Referências:

Visão geral do framework DUnitX
http://docwiki.embarcadero.com/RADStudio/XE8/en/DUnitX_Overview

Por que usar TDD
http://tableless.com.br/tdd-por-que-usar/

Introdução à Refatoração
http://www.linhadecodigo.com.br/artigo/2832/introducao-a-refatoracao.aspx

MYERS, Glenford J. The Art of Software Testing. New Jersey: John Wiley & Sons Inc., 2004.

PRESSMAN, Roger S. Engenharia de Software – Uma Abordagem Profissional. 7 ed. Porto Alegre: AMGH Editora, 2011.

RÄTZMANN, Manfred; DE YOUNG, Clinton. Galileo Computing – Software Testing and Internationalization. Salt Lake City: Lemoine International Inc., 2003.

SOMMERVILLE, Ian. Engenharia de Software. 8 ed. São Paulo: Pearson Addison-Wesley, 2007.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

REVISTAS DIGITAIS

Imagine poder ter acesso a todas as edições das revistas Java Magazine, .NET Magazine, SQL Magazine, Mobile Magazine, Engenharia de Software, ClubeDelphi e Easy Java.

São mais de 4.000 artigos publicados!

Uma verdadeira biblioteca online perfeita para seus estudos!

Para mais informações :
http://www.devmedia.com.br/mvp

DEVMEDIA

Aprendizagem de Máquina com Intelligence Lab

Aprenda a construir um aplicativo de previsão com Inteligência Artificial

A aprendizagem de máquina é uma subárea da Inteligência Artificial que visa dotar as máquinas de habilidades para aprenderem, assim como os seres humanos fazem. Por exemplo, para um aluno aprender determinados cálculos em uma aula de matemática, ele precisará fazer vários exercícios repetitivos com o intuito de praticar até conseguir atingir um bom nível de aprendizado. Um computador, por sua vez, aprende de forma bastante similar; porém, em ambientes comerciais são utilizados dados de bases em conjunto com algoritmos desenvolvidos especificamente para esse fim. Com isso, para que um computador consiga aprender alguma coisa são necessários basicamente dois componentes principais: os dados e os algoritmos que possam lidar com esses dados.

Existem diversas técnicas de aprendizagem de máquina ou mineração de dados, e cada uma delas está relacionada a diferentes tipos de informações e/ou conhecimento que podem ser gerados como resultado. Uma das principais e mais comumente utilizadas é a tarefa de classificação, que consiste em observar dados do passado para rotular novos registros que até então não se sabe a que classe pertencem. Tomando como base a **Tabela 1**, que representa os dados históricos de vinte pessoas, deseja-se prever que tipo de carro elas irão comprar com base em suas características pessoais (a próxima seção discorre sobre os campos dessa tabela). Em outras palavras, o objetivo é encontrar uma relação entre o tipo de carro que uma pessoa irá comprar com os atributos idade, renda mensal e casa própria. Baseado nisso, um problema de classificação é composto por um conjunto de atributos previsores e um atributo meta ou classe, e o objetivo é prever o valor da classe com os outros atributos.

Fique por Dentro

A tomada de decisão auxiliada por computadores tem se tornado cada vez mais comum, devido à crescente quantidade de informações gerada e demanda por resultados mais rápidos. Em cenários assim, os dados históricos das empresas têm sido utilizados como parâmetros de entrada para algoritmos inteligentes de aprendizagem de máquina, capazes de auxiliar na tomada de decisão com base em informações anteriores e regras de aprendizado. Saber como desenvolver aplicativos utilizando técnicas de Inteligência Artificial é útil porque permite a adição de funcionalidades avançadas em aplicações gerenciais, por exemplo, tornando-as auxiliares poderosos na tomada de decisão e automatização de atividades nas empresas.

Nota

Os dados da **Tabela 1** foram criados de forma aleatória e nenhuma real análise sobre esse cenário foi realizada.

É importante salientar que para que se consiga realizar uma previsão são necessários dados históricos e que a classe seja conhecida. Esse processo de conhecer previamente a classe é chamado de aprendizagem de máquina supervisionada, ou seja, existe a figura de um supervisor que irá indicar a que classe cada um dos registros da base de dados pertence. Por exemplo, o primeiro registro da **Tabela 1** é uma pessoa na faixa de 18-28 anos, que possui uma renda mensal entre 3001 e 6000, possui casa

Idade (previsor)	Renda mensal (previsor)	Casa própria (previsor)	Tipo de carro (classe)
18-28	Entre 3001 e 6000	Sim	Popular
29-39	Entre 3001 e 6000	Não	Luxo
40+	Entre 1501 e 3000	Não	Popular
40+	Até 1500	Não	Popular
29-39	Entre 1501 e 3000	Sim	Popular
18-28	Mais que 6000	Não	Luxo
29-39	Entre 3001 e 6000	Sim	Luxo
18-28	Mais que 6000	Não	Popular
40+	Entre 3001 e 6000	Sim	Popular
18-28	Até 1500	Não	Luxo
29-39	Entre 3001 e 6000	Não	Popular
40+	Até 1500	Não	Luxo
18-28	Mais que 6000	Sim	Luxo
29-39	Mais que 6000	Sim	Luxo
40+	Até 1500	Sim	Luxo
40+	Entre 1501 e 3000	Sim	Luxo
18-28	Entre 1501 e 3000	Sim	Popular
29-39	Entre 1501 e 3000	Não	Popular
40+	Mais que 6000	Sim	Popular
18-28	Entre 1501 e 3000	Sim	Popular

Tabela 1. Base de dados de exemplo

própria e adquiriu um carro popular. Assim como esse registro, todos os outros dados são provenientes do passado, e por meio deles objetiva-se encontrar algum tipo de padrão capaz de prever o comportamento de outras pessoas que possam vir a adquirir um carro no futuro. Nesse contexto, essa previsão pode ser utilizada como estratégia de negócio, ou seja: sabendo-se o perfil do cliente pode-se ofertar carros mais caros ou mais baratos. Além desse exemplo, as técnicas de classificação já têm sido utilizadas em âmbito empresarial para previsões financeiras (principalmente por bancos), venda de produtos e diversas outras áreas, independente do ramo de negócio.

Para que todo esse processo seja possível, são necessários algoritmos específicos que possam extrair os dados da base e gerar modelos preditivos, os quais possam ser utilizados posteriormente para classificar novos registros. Existem diversos tipos de paradigmas de algoritmos para essas funções, e eles podem ser baseados em matemática, estatística, árvores de decisão, redes neurais, dentre outros.

Um algoritmo bastante utilizado nesse contexto é o Naive Bayes, que é baseado em probabilidade e estatística e será o foco desse artigo. O funcionamento dele é bastante simples: analisa todos os dados e constrói uma tabela de probabilidades combinando os atributos previsores com a respectiva classe. Esse processo é chamado de treinamento, pois por meio dos dados históricos o algoritmo consegue construir o seu aprendizado, que nada mais é do que a própria tabela de probabilidades. A **Tabela 2** apresenta o resultado do treinamento do Naive Bayes, que foi gerado baseado na **Tabela 1**.

É possível observar nessa tabela que os dados foram agrupados pela classe (tipo de carro) à esquerda, e, na parte à direita estão os atributos previsores juntamente com os respectivos valores. Observa-se que a tabela está dividida basicamente em duas linhas, uma para o valor "Popular" e outra para o valor "Luxo", os quais são os objetivos da previsão. Nota-se também que dos vinte registros, onze pertencem à classe "Popular" (55%) e nove à classe "Luxo" (45%). Para melhor entender a disposição desses dados, será considerado somente o campo *Idade* da **Tabela 2**. Esse atributo possui três valores, ou seja: existem sete registros com a faixa 18-28, seis registros na faixa 29-39 e sete pessoas na faixa 40+ (os números estão indicados ao lado da faixa etária). Tomando como exemplo somente a faixa 18-28, percebe-se que dos sete registros nessa faixa etária, quatro compraram um carro popular (58%) e

Tipo de carro (20)	Idade			Renda mensal				Casa própria	
	18-28 (7)	29-39 (6)	40+ (7)	Até 1500 (4)	Entre 1501 e 3000 (6)	Entre 3001 e 6000 (5)	Mais que 6000 (5)	Sim (11)	Não (9)
Popular	4/7	3/6	4/7	1/4	5/6	3/5	2/5	6/11	5/9
11/20 55%	58%	50%	58%	25%	84%	60%	40%	55%	55%
Luxo	3/7	3/6	3/7	3/4	1/6	2/5	3/5	5/11	4/9
9/20 45%	42%	50%	42%	75%	16%	40%	60%	45%	45%

Tabela 2. Treinamento no Naive Bayes

três compraram um carro de luxo (42%); conforme pode ser visto em negrito essa coluna na tabela. O mesmo segue-se para todos os outros campos, para os quais as estimativas de probabilidade são feitas de forma similar.

Com isso já é possível realizar previsões utilizando registros que não se sabe a qual classe podem pertencer. Por exemplo, suponha-se que um novo cliente com os seguintes dados pretenda comprar um novo carro: 29-39 (idade), Até 1500 (renda mensal) e Não (casa própria). O algoritmo Naive Bayes faz as estimativas para cada uma das classes (popular ou luxo), e aquela que tiver o maior valor é considerada como decisão final. A seguir são mostrados os cálculos:

$$\text{Carro popular} = P_{popular} * \text{29-39}_{popular} * \text{Até 1500}_{popular} * \text{Não}_{popular}$$
$$\text{Carro popular} = 11/20 * 3/6 * 1/4 * 5/9$$
$$\text{Carro popular} = 0{,}038$$

$$\text{Carro luxo} = P_{luxo} * \text{29-39}_{luxo} * \text{Até 1500}_{luxo} * \text{Não}_{luxo}$$
$$\text{Carro luxo} = 9/20 * 3/6 * 3/4 * 4/9$$
$$\text{Carro luxo} = 0{,}075$$

Para calcular a previsão para a classe carro popular, é necessário fazer a multiplicação de cada um dos valores, ou seja: a probabilidade de ser comprado um carro popular (11/20), a probabilidade de estar na faixa 29-39 e ter comprado um carro popular (3/6), a probabilidade de ter uma renda até 1500 e ter comprado um carro popular (1/4) e por fim, a probabilidade de não ter casa própria e ter comprado um carro popular (5/9). O mesmo ocorre para o cálculo da classe carro de luxo, com a diferença de que são utilizadas as probabilidades referentes a essa classe. É importante salientar que esses valores são extraídos da **Tabela 2**, e como conclusão, uma pessoa com essas características tem mais chances de comprar um carro de luxo, visto que 0,075 é maior do que 0,038. Dessa forma, esse algoritmo de aprendizagem de máquina utiliza um processo de indução que objetiva analisar dados específicos para posteriormente generalizar registros cuja classe não é conhecida.

Com base nesses conceitos, o objetivo desse artigo é mostrar como construir um aplicativo móvel no Delphi 10 Seattle que faça toda a operacionalização desse processo de aprendizagem de máquina, utilizando para isso o algoritmo Naive Bayes da biblioteca Intelligence Lab. Essa biblioteca é de propriedade da empresa norte-americana Mitov Software e apresenta uma suíte de componentes específicos para Inteligência Artificial no Delphi, possuindo recursos para implementação de agrupamento de dados, filtros de SPAM, visão computacional, reconhecimento de voz, aplicações OCR (*Optical Character Recognition*, ou reconhecimento de texto em imagens) e praticamente todo tipo de tarefa de classificação. Uma característica importante dessa biblioteca é que ela possui versão para o FireMonkey, o que possibilita a construção de aplicativos móveis inteligentes, fato sobre o qual muito se tem ouvido muito falar no mercado: a integração entre dispositivos móveis e técnicas de Inteligência Artificial. Por este motivo, optou-se por abordar nesse artigo como construir um pequeno exemplo de aplicativo móvel rodando em um dispositivo Android. As próximas seções apresentam todas as etapas, iniciando com a criação da base de dados e da interface gráfica para depois abordar os procedimentos para a implementação da aprendizagem de máquina.

Nota

Os valores retornados pela previsão não estão em porcentagem, neste caso, e para fazer essa conversão basta somar os dois resultados e dividir o valor predito pela soma. Por exemplo: 0,038 + 0,075 = 0,113 – Carro popular: (0,038 / 0,113) * 100 = 33,62% – Carro luxo: (0,075 / 0,113) * 100 = 66,38%. Neste ponto, porém, esse cálculo não foi necessário, pois é possível identificar os resultados avaliando apenas os valores na forma como foram retornados.

Nota

Visite a seção **Links** para ter acesso ao site oficial da biblioteca Intelligence Lab da Mitov Software. A Intelligence Lab possui uma versão totalmente funcional (sem código fonte) para uso acadêmico e não comercial.

Criando a base de dados e configurando o acesso aos dados

O primeiro passo é a criação de uma nova aplicação multidispositivo no RAD Studio (*File>New>Multi-Device Application – Delphi*), e logo após a definição da base de dados utilizando o SQLite, que

é uma forma comumente utilizada para a manipulação de bases de dados em aplicativos móveis.

Para isso, será utilizado o Data Explorer do RAD Studio (*View>Data Explorer*), no qual deve ser adicionada uma nova conexão em *FireDAC>SQLite database* chamada *naivebayes* (botão direito em *SQLite database* e *Add new connection*). Na janela *FireDAC Connection Editor* as únicas propriedades que devem ser configuradas são a *Database* e a *LockingMode*. Na primeira deve ser salvo o arquivo com a extensão *sdb* ou *db*, que representa a base de dados e seu respectivo caminho. A segunda propriedade deve ser configurada com o valor *Normal* para que seja possível realizar alterações na base de dados. Nessa mesma janela, na guia *SqlScript* o comando SQL da **Listagem 1** deve ser executado, o qual tem a função de criar a tabela com os dados das pessoas.

Listagem 1. SQL para criação da tabela de pessoas.

```
1 create table pessoas (
2  idpessoa integer primary key autoincrement,
3  nome varchar(50) not null,
4  idade varchar(10) not null,
5  renda_mensal varchar (20) not null,
6  casa_propria char(3) not null,
7  tipo_carro varchar(10) not null
8 );
```

Essa tabela possui seis campos, sendo três deles os atributos previsores (*idade*, *renda_mensal* e *casa_propria*) e um a classe (*tipo_carro*). Dessa forma, objetiva-se prever qual o tipo de carro uma determinada pessoa irá comprar com base em suas características. Pode-se observar que esses quatro campos são do tipo texto e todos eles representam faixas de valores, por exemplo: a idade é composta pelas faixas "18-28", "29-39" e "40+" e a renda mensal pelos valores "Até 1500", "Entre 1501 e 3000", "Entre 3001 e 6000" e "Mais que 6000". Por outro lado, a casa própria é composta somente pelos valores "Sim" e "Não" e por fim, o tipo do carro possui os valores "Popular" e "Luxo". É importante salientar que as colunas *idpessoa* e *nome* não entram nas análises preditivas pelo fato de serem campos únicos e que não representam agregações. Por exemplo, o *idpessoa* é a chave primária e cada registro possui um valor diferente, enquanto que o campo *nome* não representa um dado importante para realizar previsões, ou seja, o algoritmo não irá encontrar generalizações do tipo: pessoas com o nome Maria tem mais chances de comprar carros de luxo. Em tarefas preditivas como a apresentada nesse exemplo, busca-se atributos que possam diferenciar os indivíduos.

Nota

A tabela de pessoas já está em um formato específico para que as previsões sejam feitas, bem como os valores dos campos já estão distribuídos em faixas. Em uma base de dados transacional, várias conversões e acesso a outras tabelas seriam necessárias para que esses dados fossem transformados nesse formato.

A **Listagem 2** apresenta os comandos SQL para a inserção de vinte pessoas, os quais também devem ser executados na guia *SqlScript* da janela *FireDAC Connection Editor*, conforme apresentado na **Figura 1**. Pode-se observar que como a chave primária é autoincremento, não foi necessário passar essa coluna como parâmetro para a instrução SQL. Além disso, os dados inseridos são os mesmos apresentados na **Tabela 1**.

Listagem 2. SQL para inserção das pessoas.

```
1 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('João', '18-28', 'Entre 3001 e 6000', 'Sim', 'Popular');
2 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Maria', '29-39', 'Entre 3001 e 6000', 'Não', 'Luxo');
3 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Carlos', '40+', 'Entre 1501 e 3000', 'Não', 'Popular');
4 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('José', '40+', 'Até 1500', 'Não', 'Popular');
5 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Júlia', '29-39', 'Entre 1501 e 3000', 'Sim', 'Popular');
6 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Ana', '18-28', 'Mais que 6000', 'Não', 'Luxo');
7 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Jones', '29-39', 'Entre 3001 e 6000', 'Sim', 'Luxo');
8 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Kelvin', '18-28', 'Mais que 6000', 'Não', 'Popular');
9 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Leandro', '40+', 'Entre 3001 e 6000', 'Sim', 'Popular');
10 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Carolina', '18-28', 'Até 1500', 'Não', 'Luxo');
11 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('William', '29-39', 'Entre 3001 e 6000', 'Não', 'Popular');
12 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Alan', '40+', 'Até 1500', 'Não', 'Luxo');
13 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Roni', '18-28', 'Mais que 6000', 'Sim', 'Luxo');
14 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('João Paulo', '29-39', 'Mais que 6000', 'Sim', 'Luxo');
15 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Alisson', '40+', 'Até 1500', 'Sim', 'Luxo');
16 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Monica', '40+', 'Entre 1501 e 3000', 'Sim', 'Luxo');
17 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Tatiane', '18-28', 'Entre 1501 e 3000', 'Sim', 'Popular');
18 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Rebeca', '29-39', 'Entre 1501 e 3000', 'Não', 'Popular');
19 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Milene', '40+', 'Mais que 6000', 'Sim', 'Popular');
20 insert into pessoas (nome, idade, renda_mensal, casa_propria, tipo_carro)
   values ('Larissa', '18-28', 'Entre 1501 e 3000', 'Sim', 'Popular');
```

Após a base de dados estar concluída, é necessário criar um novo Data Module (*dmConexao*) para que os componentes de acesso a dados fiquem centralizados (*File>New>Other>Delphi files>Data Module*). Cinco componentes devem ser adicionados: um TFDConnection (*conexao*), dois TFDQuery (*consulta* e *trei-*

namento), um TFDGUIxWaitCursor (*cursor*) e um TFDPhysisSQLiteDriverLink (*driver*). Um duplo clique no componente *conexao* abre novamente o *FireDAC Connection Editor*, que deve ter o combobox *Drive ID* configurado para o *SQLite*. As propriedades *Database* e *LockingMode* devem ter os mesmos valores que foram configurados anteriormente, enquanto que a propriedade *LoginPrompt* deve ter seu valor configurado para *False* para que a senha do banco de dados não seja solicitada a cada conexão. O componente *consulta* deve ter sua propriedade *Connection* setada para o componente *conexao* e na propriedade *SQL* a seguinte instrução deve ser inserida:

```
select nome, idade from pessoas
```

Esse comando retorna somente o nome e a idade das pessoas, os quais serão posteriormente adicionados no ListView. Além disso, faz-se necessário adicionar os campos na consulta: botão direito no componente *consulta* e *Add all fields*. O componente *treinamento* deve ter sua propriedade *Connection* também setada para *conexao*, sendo utilizado depois para construir a base de treinamento. Por fim, os componentes *cursor* e *driver* não precisam ter nenhuma propriedade configurada, e ambos são utilizados para gerenciar o cursor dos dados na interface gráfica e a ligação do driver para o SQLite. A **Figura 2** apresenta o Data Module com todos os componentes adicionados.

É ainda necessário programar o evento *OnCreate* do Data Module, conforme mostrado na **Listagem 3**. Esse código executa o comando SQL do TFDQuery e carrega para a memória todos os registros da tabela *pessoas*.

Criando a interface gráfica e listando os registros

Como se pode observar na **Figura 3**, a interface gráfica do aplicativo é composta por duas guias. A primeira terá a função de somente apresentar os registros cadastrados na base de dados, enquanto que na segunda o usuário será capaz de fazer o treinamento do algoritmo, o

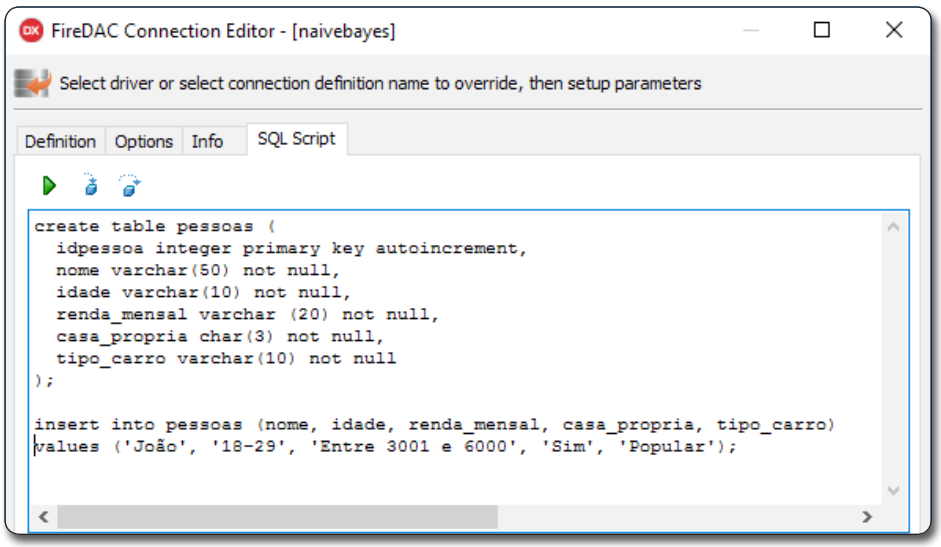


Figura 1. Execução de SQL no FireDAC Connection Editor

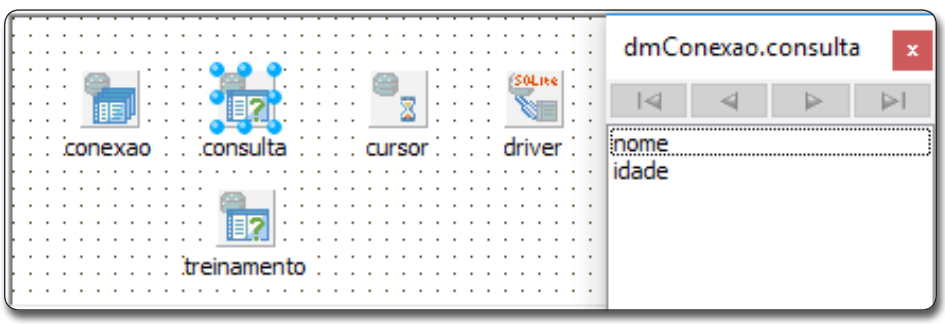


Figura 2. Configuração do Data Module

Listagem 3. OnCreate do Data Module.

```
1 procedure TdmConexao.DataModuleCreate(Sender: TObject);
2 begin
3   consulta.Open();
4 end;
```

preenchimento dos dados de uma nova pessoa a ser classificada e, por fim, prever a qual classe essa pessoa irá pertencer. Para construir essa janela, inicialmente deve ser adicionado um TTabControl (*tabPrincipal*) com a propriedade *Align* configurada para *Client*, para que o componente preencha toda a janela. Dentro dele devem ser adicionados dois TTabItem (*tabListagem* e *tabPrevisao*), os quais devem ter sua propriedade *Text* alteradas para *Listagem* e *Previsão*, respectivamente. Na primeira guia deve ser adicionado um TListView (*lswListagem*), enquanto que a segunda guia deve conter quatro TLabel (*lblIdade*, *lblRendaMensal*, *lblCasaPropria* e *lblResultado*), três TComboBox (*cmbIdade*, *cmbRendaMensal* e *cmbCasaPropria*) e dois TButton (*btnTreinar* e *btnPrever*), conforme mostrado à direita da **Figura 3**. Os componentes TComboBox necessitam listar os dados que correspondem às faixas de valores explanados na seção anterior, ou seja, a propriedade *Items* de cada um deve ser preenchida. Por exemplo, para o *cmbIdade* as seguintes faixas precisam ser adicionadas: "18-28", "29-39" e "40+" (o mesmo procedimento deve ser seguido para os demais componentes). Por fim, deve ser também adicionado um componente TILNaiveBayes (*naivebayes*) da paleta Intelligence Lab. Este componente será o responsável por receber os dados do TFDQuery, realizar o treinamento por

meio da construção da tabela de probabilidades e realizar as predições conforme os dados informados pelo usuário.

O próximo passo é listar os registros do TFDQuery (*consulta*) no ListView (*lswListagem*), e para isso, primeiramente é necessário configurar a propriedade *ItemAppearance>ItemAppearance* para *ListItemRightDetail*, para que seja possível listar um texto e um detalhe no ListView. A propriedade *ItemAppearanceObjects>ItemObjects>Acessory>Visible* deve ser marcada para *False*, para que assim nenhum botão de mais detalhes seja exibido ao lado de cada registro. Por fim, a **Figura 4** apresenta as ligações de Live Bindings necessárias para que a listagem dos registros seja concluída. Conforme se pode observar, o campo *nome* é ligado ao *Item.Text* e o campo *idade* ao *Item.Detail*. O asterisco (*) é utilizado para sincronizar os componentes visuais com a lista.

Figura 3. Interface gráfica do aplicativo

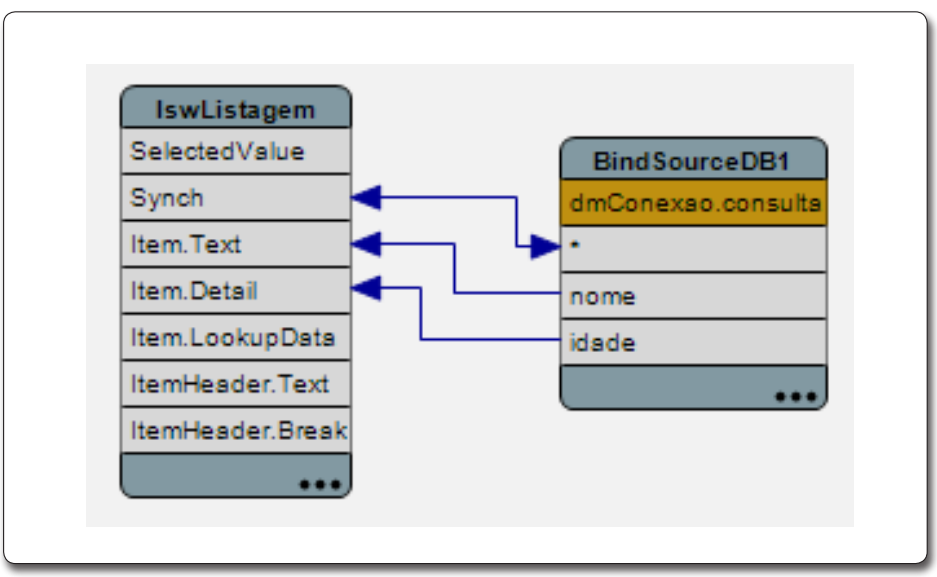


Figura 4. Live Bindings para listagem dos registros.

Treinando o Naive Bayes com os dados

Bibliotecas de aprendizagem de máquina, como o Weka para Java, ou os pacotes para Python, possibilitam que sejam fornecidos como parâmetro os dados em forma categórica, ou seja, em formato texto conforme foi apresentado anteriormente. Porém, o Intelligence Lab trabalha com matrizes de números reais, ou seja, não é possível passar como parâmetro os dados em forma textual como eles estão nesse momento. Portanto, é necessário realizar uma conversão nesses dados para torná-los compatíveis. A **Tabela 3** apresenta o resultado dessa conversão, sendo possível verificar para cada campo os valores originais na segunda coluna e os valores numéricos convertidos na terceira. Percebe-se que foram utilizados valores relativos a cada uma das faixas, no intervalo entre 0.00 e 1.00.

Nota

Para que a paleta Intelligence Lab esteja disponível na IDE do RAD Studio, deve ser feito o download do executável de instalação pelo site oficial da empresa Mitov Software. O processo de instalação é bastante simples e consiste somente de seguir os passos do instalador sem nenhuma configuração adicional. A versão utilizada nesse artigo foi a 7.8.2.

Nota

Antes de utilizar a opção Bindg Visually do RAD Studio, a unit do Data Module deve ser adicionada no formulário principal (File>Use Unit). Após a ligação via Live Bindings, serão adicionados automaticamente no formulário dois novos componentes: BindingsList1 e BindSourceDB1.

Como resultado, a **Tabela 4** apresenta o conjunto final dos valores que serão efetivamente utilizados para o treinamento do algoritmo Naive Bayes. Todo o processo apresentado anteriormente sobre a geração da tabela de probabilidades é o mesmo, a única diferença é a forma como os dados serão apresentados. Em outras palavras, ao invés das colunas da **Tabela 2** serem representadas pelos valores textuais, serão mostrados na forma numérica.

Como os dados da tabela de pessoas estão no formato categórico, é necessário criar funções no aplicativo que façam esse mapeamento, ou seja, que recebam como

parâmetro um texto e retornem um número real correspondente, conforme a **Tabela 3**. Para isso, a **Listagem 4** apresenta a assinatura das quatro funções (uma para cada campo) que farão o mapeamento, as quais devem ser declaradas na seção *private* do formulário. Para gerar a implementação, deve-se apertar as teclas *CTRL+SHIFT+C*.

Nota

Uma maneira alternativa seria armazenar os valores na base de dados já no formato numérico, não sendo necessárias codificações adicionais para realizar a conversão dos dados. Por fins didáticos e um melhor entendimento do algoritmo Naive Bayes, nesse exemplo foi trabalhado inicialmente com as faixas de valores do tipo textuais.

Campo	Valor original	Valor convertido
Idade	18-28	0.00
	29-39	0.50
	40+	1.00
Renda mensal	Até 1500	0.00
	Entre 1501 e 3000	0.50
	Entre 3001 e 6000	0.75
	Mais que 6000	1.00
Casa própria	Sim	1.00
	Não	0.00
Tipo de carro	Popular	0.00
	Luxo	1.00

Tabela 3. Conversão das faixas textuais em valores numéricos

Idade (previsor)	Renda mensal (previsor)	Casa própria (previsor)	Tipo de carro (classe)
0.00	0.75	1.00	0.00
0.50	0.75	0.00	1.00
1.00	0.50	0.00	0.00
1.00	0.00	0.00	0.00
0.50	0.50	1.00	0.00
0.00	1.00	0.00	1.00
0.50	0.75	1.00	1.00
0.00	1.00	0.00	0.00
1.00	0.75	1.00	0.00
0.00	0.00	0.00	1.00
0.50	0.75	0.00	0.00
1.00	0.00	0.00	1.00
0.00	1.00	1.00	1.00
0.50	1.00	1.00	1.00
1.00	0.00	1.00	1.00
1.00	0.50	1.00	1.00
0.00	0.50	1.00	0.00
0.50	0.50	0.00	0.00
1.00	1.00	1.00	0.00
0.00	0.50	1.00	0.00

Tabela 4. Dados de treinamento em formato numérico

Listagem 4. Definição das funções para mapeamento.

```
1 private
2  function GetIdadeNumerico(Idade: string): Real;
3  function GetRendaMensalNumerico(RendaMensal: string): Real;
4  function GetCasaPropriaNumerico(CasaPropria: string): Real;
5  function GetTipoCarroNumerico(TipoCarro: string): Real;
6 public
7 end;
```

A **Listagem 5** mostra as quatro funções implementadas. Entre as linhas 1 e 9 é mostrada a função para converter a idade, a qual recebe como parâmetro um tipo de dado string, faz as comparações conforme a **Tabela 3** e retorna o resultado utilizando o tipo de dados *real*. Entre as linhas 10 e 20 é mostrada a função para converter a renda mensal, entre as linhas 21 e 27 a conversão do campo casa própria e, por fim, entre as linhas 28 e 34 a conversão do tipo de carro.

Listagem 5. Implementação das funções de mapeamento.

```
1 function TfrmPrincipal.GetIdadeNumerico(Idade: string): Real;
2 begin
3  if Idade = '18-28' then
4   Result := 0.00
5  else if Idade = '29-39' then
6   Result := 0.50
7  else if Idade = '40+' then
8   Result := 1.00;
9 end;
10 function TfrmPrincipal.GetRendaMensalNumerico(RendaMensal: string): Real;
11 begin
12  if RendaMensal = 'Até 1500' then
13   Result := 0.00
14  else if RendaMensal = 'Entre 1501 e 3000' then
15   Result := 0.500
16  else if RendaMensal = 'Entre 3001 e 6000' then
17   Result := 0.750
18  else if RendaMensal = 'Mais que 6000' then
19   Result := 1.00;
20 end;
21 function TfrmPrincipal.GetCasaPropriaNumerico(CasaPropria: string): Real;
22 begin
23  if CasaPropria = 'Não' then
24   Result := 0.00
25  else if CasaPropria = 'Sim' then
26   Result := 1.00;
27 end;
28 function TfrmPrincipal.GetTipoCarroNumerico(TipoCarro: string): Real;
29 begin
30  if TipoCarro = 'Popular' then
31   Result := 0.00
32  else if TipoCarro = 'Luxo' then
33   Result := 1.00;
34 end;
```

A **Listagem 6** apresenta o código necessário para efetivamente realizar o treinamento utilizando os dados das pessoas armazenadas na base de dados, o qual deve ser informado no evento *OnClick* do *btnTreinar*. Na linha 3 é declarada uma variável do tipo *ISLRealMatrixBuffer*, a qual tem a função de armazenar toda a matriz com os dados, enquanto que na linha 4 é declarada a variável *ISLRealBuffer* que conterá os dados referentes ao atributo meta. Como pode ser observado, o Intelligence Lab trabalha com essas duas variáveis para armazenar todo o conjunto de dados, ou seja, a primeira é utilizada para armazenamento somente dos atributos previsores (idade, renda mensal e casa própria), enquanto que a segunda diz respeito somente ao atributo meta ou à classe (tipo do carro). Na linha 8 a variável *ALinhas* recebe a quantidade de registros existentes na tabela de pessoas (20 nesse exemplo), para que posteriormente esse valor sirva de parâmetro de entrada para a construção da matriz de dados. Similarmente, na linha 9 a variável *AColunas* recebe a quantidade dos campos previsores, ou seja, o número 3. Na linha 10 os dados de treinamento são criados passando como parâmetro para o construtor as variáveis *ALinhas* e *AColunas*, que possui respectivamente os valores 20 e 3. Dessa forma, está sendo criada uma matriz de números reais com vinte linhas (quantidade de registros) e três colunas (quantidade de campos previsores). Similarmente, na linha 11 a variável *ARespostas* representa um vetor também com vinte posições, nas quais serão armazenados os atributos classe para cada registro. Em resumo, levando-se em conta os dados da **Tabela 4**, a variável *ADadosTreinamento* armazena os atributos previsores, enquanto que a variável *ARespostas* mantém o atributo meta.

Na linha 12, a variável *ALinhaAtual* tem a função de controlar a repetição das linhas, variando de 0 até a quantidade de registros pertencentes a cada classe. Entre as linhas 13 e 22 encontra-se o código para ler e armazenar na matriz somente os registros de pessoas que pertencem à classe "Popular" no que se refere ao tipo de carro. Por outro lado, entre as linhas 25 e 34 se dá o processo de leitura e armazenamento somente das pessoas pertencentes à classe carro de "Luxo". Essa separação na ordem para cadastrar os dados é necessária devido ao funcionamento interno da classe *ISLRealMatrixBuffer*, que necessita que os registros da mesma classe sejam importados em sequência (erros de ponteiros são gerados caso essa ordem não seja respeitada).

Listagem 6. Treinamento do Naive Bayes.

```
1 procedure TfrmPrincipal.btnTreinarClick(Sender: TObject);
2 var
3   ADadosTreinamento: ISLRealMatrixBuffer;
4   ARespostas: ISLRealBuffer;
5   ALinhas, AColunas: Integer;
6   ALinhaAtual: Integer;
7 begin
8   ALinhas := dmConexao.consulta.RecordCount;
9   AColunas := 3;
10  ADadosTreinamento := TSLRealMatrixBuffer.CreateSize(ALinhas, AColunas);
11  ARespostas := TSLRealBuffer.CreateSize(ALinhas);
12  ALinhaAtual := 0;
13  dmConexao.treinamento.Open('select idade, renda_mensal, casa_propria,
      tipo_carro from pessoas where tipo_carro = ''Popular''');
14  while not dmConexao.treinamento.Eof do
15  begin
16    ARespostas[ALinhaAtual] := GetTipoCarroNumerico(dmConexao.treinamento.
      FieldByName('tipo_carro').AsString);
17    ADadosTreinamento[ALinhaAtual, 0] := GetIdadeNumerico(dmConexao.
       treinamento.FieldByName('idade').AsString);
18    ADadosTreinamento[ALinhaAtual, 1] := GetRendaMensalNumerico
       (dmConexao.treinamento.FieldByName('renda_mensal').AsString);
19    ADadosTreinamento[ALinhaAtual, 2] := GetCasaPropriaNumerico
      (dmConexao.treinamento.FieldByName('casa_propria').AsString);
20    Inc(ALinhaAtual);
21    dmConexao.treinamento.Next;
22  end;
23  ALinhaAtual := 0;
24  dmConexao.treinamento.Close;
25  dmConexao.treinamento.Open('select idade, renda_mensal, casa_propria,
      tipo_carro from pessoas where tipo_carro = ''Luxo''');
26  while not dmConexao.treinamento.Eof do
27  begin
28    ARespostas[ALinhaAtual] := GetTipoCarroNumerico(dmConexao.treinamento.
      FieldByName('tipo_carro').AsString);
29    ADadosTreinamento[ALinhaAtual, 0] := GetIdadeNumerico
      (dmConexao.treinamento.FieldByName('idade').AsString);
30    ADadosTreinamento[ALinhaAtual, 1] := GetRendaMensalNumerico
      (dmConexao.treinamento.FieldByName('renda_mensal').AsString);
31    ADadosTreinamento[ALinhaAtual, 2] := GetCasaPropriaNumerico
       (dmConexao.treinamento.FieldByName('casa_propria').AsString);
32    Inc(ALinhaAtual);
33    dmConexao.treinamento.Next;
34  end;
35  naivebayes.Train(ADadosTreinamento, ARespostas, False);
36 end;
```

Com isso, na linha 13 é feita uma consulta somente para recuperar da base de dados os registros da classe "Popular", como pode ser visto no comando *where*. Na linha 14 é definida uma estrutura de repetição para percorrer registro por registro, e em seguida eles são cadastrados na base de treinamento. Na linha 16 é preenchido na variável *ARespostas* o atributo meta, invocando para isso o método *GetTipoCarroNumerico* que recebe como parâmetro o valor em formato de texto e retorna o valor da classe ("0" para carro popular). Entre as linhas 17 e 19 são copiados para cada coluna da matriz os dados das pessoas (colunas 0, 1 e 2), sendo primeiramente convertidos por meio das funções definidas na **Listagem 5**. Na linha 23 a variável *ALinhaAtual* é zerada, pois quando os dados de uma nova classe devem ser preenchidos, o contador precisa ser reiniciado com o valor zero novamente. A linha 25 apresenta o novo comando SQL para buscar somente as pessoas que estão na classe carros de luxo, e entre as linhas 26 e 34 o mesmo procedimento é executado.

Por fim, na linha 35 é invocado o método *Train* do componente *TILNaiveBayes*, que recebe como parâmetro os dados de treinamento juntamente com as respostas. Esse método é o responsável por efetivamente criar a tabela de probabilidades vista em detalhes anteriormente, para que então as previsões possam ser realizadas. O terceiro parâmetro deste método (*AUpdate*) é referente à atualização da base de treinamento, ou seja: se o valor for *True* a base de treinamento é adicionada à que já existe, e por outro lado, se o valor for *False* a base antiga é apagada e o classificador é treinado novamente do zero. Na **Listagem 6** esse parâmetro recebe o valor *False* para que a cada clique no botão *btnTreinar* uma nova tabela de probabilidades seja gerada. Caso o valor fosse *True*, a cada vez que o botão treinar for executado os dados serão acrescentados para formar uma nova tabela combinando todos eles (por exemplo, clicando três vezes no botão será gerada uma tabela com 60 registros).

Nota

Na versão utilizada nesse exemplo (7.8.2), o componente TILNaiveBayes não apresenta nenhum método que mostre em tela a tabela de probabilidades gerada.

Quando o treinamento já está pronto, basta utilizar a tabela de probabilidades gerada pelo Naive Bayes para classificar novos registros, assim como foi abordado de forma manual no início do artigo. Para isso, o código apresentado na **Listagem 7** deve ser escrito no evento *OnClick* do *btnPrevisao*.

Listagem 7. Passando os dados para previsão.

```
1 procedure TfrmPrincipal.btnPrevisaoClick(Sender: TObject);
2 var
3  ADadosTeste: ISLRealBuffer;
4  AIdade, ARendaMensal, ACasaPropria: real;
5 begin
6  AIdade := GetIdadeNumerico(cmbIdade.Selected.Text);
7  ARendaMensal := GetRendaMensalNumerico(cmbRendaMensal.Selected.Text);
8  ACasaPropria := GetCasaPropriaNumerico(cmbCasaPropria.Selected.Text);

9  ADadosTeste := TSLRealBuffer.CreateSize(3);
10  ADadosTeste[0] := AIdade;
11  ADadosTeste[1] := ARendaMensal;
12  ADadosTeste[2] := ACasaPropria;

13  naivebayes.Predict(ADadosTeste);
14 end;
```

Na linha 7 é declarada a variável *ADadosTeste* do tipo *ISLRealBuffer*, a qual receberá os parâmetros informados pelo usuário. A linha 4, por sua vez, apresenta os campos relacionados aos atributos previsores, todos do tipo *real*; enquanto que entre as linhas 6 e 8 essas variáveis recebem os valores numéricos utilizando as funções definidas na **Listagem 5**. Com isso, as funções para conversão de caractere para numérico recebem como parâmetro o texto que o usuário selecionou em cada TComboBox. Na linha 9, a variável *ADadosTeste* é instanciada com três posições (0, 1 e 2), ou seja, uma para cada variável previsora. É importante salientar nesse momento que o campo *tipo_carro* não é passado como parâmetro porque ele se trata do atributo meta, ou seja, é este o atributo cujo valor se pretende descobrir por meio dos outros três (utilizando a tabela de probabilidade gerada no treinamento). Entre as linhas 10 e 12 os parâmetros são passados e finalmente na linha 13 a previsão é feita por meio do método *Predict*, ou seja, é por meio deste método que o algoritmo Naive Bayes submeterá os dados informados pelo usuário para a tabela de probabilidades, fará as estimativas percentuais para cada classe e informará a qual classe o registro pertence.

Quando o método *Predict* é invocado, é disparado o evento *OnResult* no componente TILNaiveBayes, o qual deve ser programado conforme apresentado na **Listagem 8**. É nesse evento que o resultado da classificação é obtido e as ações correspondentes devem ser efetuadas. Na linha 3 é acessada a propriedade *ResultClass* do parâmetro *AResult* para obter o valor da previsão, sendo feito somente uma estrutura condicional que verifica a qual classe o usuário pertence, preenchendo o TLabel *lblResultado* com os valores "Carro popular" ou "Carro de luxo". A **Figura 5** apresenta o aplicativo rodando em um dispositivo Android, sendo possível observar à esquerda a listagem dos dados e a tela de previsão à direita.

Nota

Na versão utilizada nesse exemplo, o componente TILNaiveBayes somente apresenta o resultado final da predição, ou seja, não é possível obter o valor da porcentagem que cada uma das classes obteve.

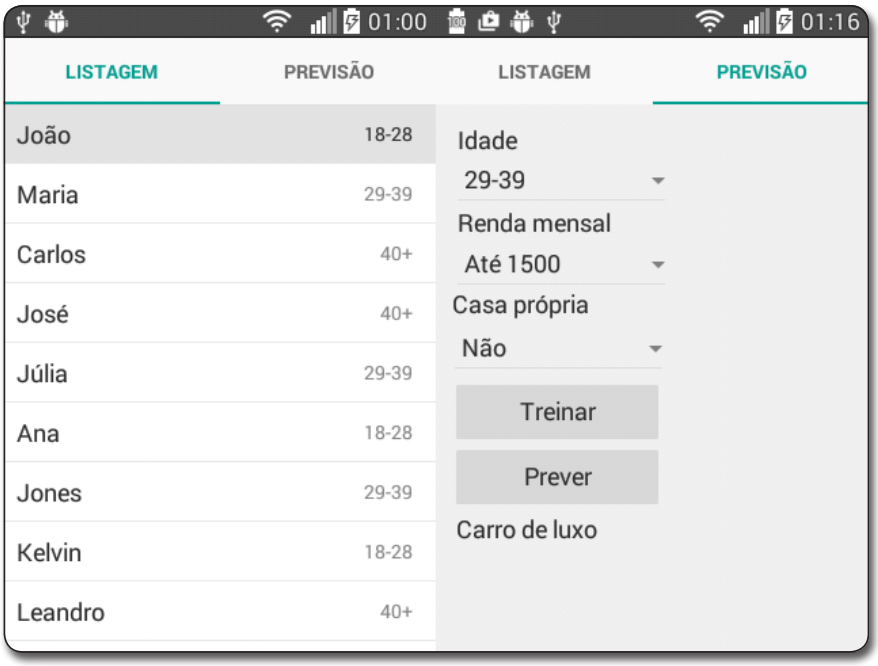


Figura 5. Aplicativo rodando em dispositivo Android

Com isso, o aplicativo móvel para realizar previsões de que tipo de carro é mais adequado para clientes de acordo com suas características está concluído.

A **Figura 6** apresenta as duas fases que foram utilizadas para a construção deste software com aprendizagem de máquina. Conforme pode ser observado, a fase 1 teve o objetivo de construir um classificador utilizando um sistema de aprendizado (Naive Bayes), o qual foi possível por meio de um conjunto de exemplos. A fase 1 corresponde ao código do evento *OnClick* do *btnTreinar*. Por outro lado, a fase 2 consistiu de submeter ao classificador um registro com classe desconhecida, para que assim uma decisão possa ser tomada com base na resposta do classificador.

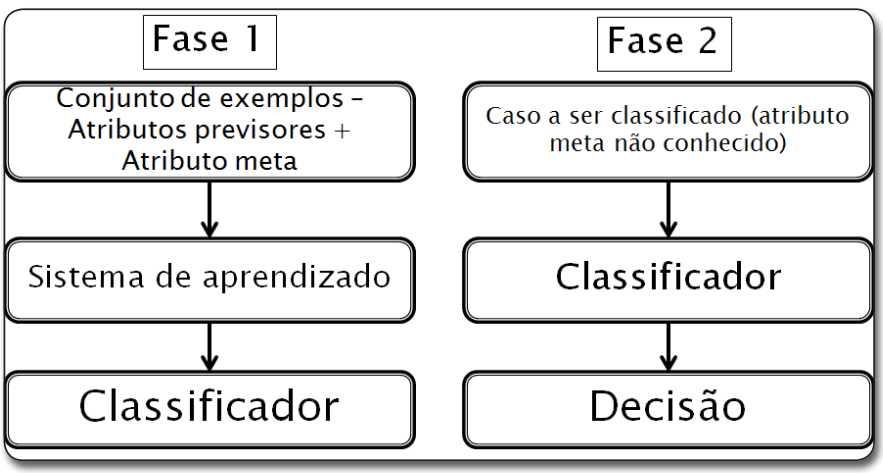


Figura 6. Fases do processo de aprendizagem de máquina

O algoritmo Naive Bayes realiza as previsões com base em probabilidade e estatística, dessa forma, a tendência é que os resultados sejam melhores de acordo com a quantidade de dados históricos que estiverem disponíveis. Com relação à base de dados, é importante frisar que os dados podem apresentar características diferentes, ou seja, as vendas de 2015 podem ser totalmente diferentes das vendas de 2016, por exemplo. Nesse caso, é necessário realizar uma análise prévia nas características dos dados antes de construir o sistema de previsão, pois os resultados podem ser "viciados" de acordo com cada ano nesse exemplo.

Outro fator não mencionado é que antes de um classificador ser utilizado comercialmente, ele precisa ser testado com vários algoritmos e validado por métodos estatísticos. Para isso, é necessário construir uma base de dados de treinamento e outra base de dados de teste, ambas com valores diferentes e com o valor da classe previamente conhecido (aprendizagem supervisionada).

Listagem 8. Obtendo o resultado da predição.

```
1 procedure TfrmPrincipal.naivebayesResult(ASender: TObject;
    AFeatures: ISLRealBuffer; AResult: TILNaiveBayesResult);
2 begin
3   if AResult.ResultClass = 0 then
4     lblResultado.Text := 'Carro popular'
5   else if AResult.ResultClass = 1 then
6     lblResultado.Text := 'Carro de luxo';
7 end;
```

Essas duas bases de dados devem conter uma quantidade razoável de registros, para que então os percentuais de acertos e erros sejam obtidos com o intuito de validar o classificador. Nesse contexto, a base de treinamento é utilizada para construir a tabela de probabilidades (Naive Bayes), enquanto que a base de teste é

usada para validar os acertos e erros. Buscam-se altos percentuais de acerto e o valor ideal depende muito do tipo de aplicação, por exemplo: em um sistema médico o percentual de acerto do sistema pode estar na faixa de 99%, visto que se o sistema indicar um remédio errado o paciente poderá ter efeitos colaterais.

O exemplo desenvolvido apresenta um botão somente para realizar o treinamento. Em um sistema real, o processo de treinamento deve estar armazenado em memória, em disco ou então ser executado na inicialização do software, somente uma vez. Isso é recomendável por basicamente dois motivos: o primeiro é que dependendo da quantidade de registros e do algoritmo utilizado, o treinamento pode demorar horas para ser finalizado, o que atrapalhará a experiência do usuário. O segundo motivo é que dados utilizados para mineração geralmente são estáticos e não dinâmicos, ou seja, por se tratarem de dados históricos, seus valores raramente são alterados e não há necessidade de treiná-los a todo momento.

Por fim, outro recurso que pode ser acrescentado nesse aplicativo é uma nova guia de cadastro para que o usuário possa incluir novas pessoas. Este recurso pode ser particularmente interessante, visto que como são poucos dados, a inclusão de alguns poucos registros pode mudar o valor das previsões, já que a cada clique no botão de treinamento a tabela de probabilidades é gerada novamente. Baseado nesse conceito é que surgiu a definição da área de aprendizagem de máquina que diz que o treinamento é obtido a partir da experiência, sendo essa obtida a partir dos dados.

Autor

Jones Granatyr
Doutorando em Informática e Mestre em Ciência da Computação, ambos na área de Inteligência Artificial. Trabalha em projetos de pesquisa relacionados à área de Inteligência Artificial, tais como Sistemas Especialistas, Mineração de Dados, Mineração de Textos e Sistemas Multiagente.

Referências e Links:

J. Granatyr, E. E. Scalabrin, F. A. Taffe, A. D. Darold, "Business Intelligence utilizando o Fast Cube", Clube Delphi, v. 166, 2016.

J. Granatyr, "Transformação de bases de dados transacionais em Data Warehouse", Clube Delphi, v. 167, 2016.

R. Angelantonio, "Processo de modelagem de Data Warehouse", SQL Magazine, v. 131, 2015.

R. Goldschmidt, E. Passos, "Data Mining: um guia prático", Editora Campus, 2005.

E. C. Gonçalves, "Naive Bayes: Mineração de dados na prática", SQL Magazine v. 110, 2013.

Intelligence Lab
http://www.mitov.com/products/intelligencelab#overview

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Programador Java: Por onde começar?

Descubra nesse vídeo como entrar na carreira Java com o pé direito!

DEVMEDIA

http://www.devmedia.com.br/programador-java-por-onde-comecar/33638

Construindo serviços RESTful com DataSnap

Exponha serviços através do HTTP com REST

Atualmente duas características têm se destacado e se mostrado extremamente importantes para grandes sistemas de software, tendo implicações diretas no sucesso dessas aplicações. A primeira delas, a escalabilidade, é a capacidade de se adequar de forma eficiente a diferentes demandas de uso. Ou seja, o sistema que atende normalmente um pequeno grupo de usuários deve estar apto a suprir um maior número de clientes sem que grande esforço seja necessário por parte da equipe de desenvolvimento. A segunda característica é a interoperabilidade, que indica a capacidade de um sistema de se comunicar, de forma transparente, com outros cuja estrutura não necessariamente é semelhante à sua. Por exemplo, uma aplicação desenvolvida em Delphi deve ser capaz de se comunicar com outra baseada em C# ou Java. Para que essa característica seja obtida com sucesso, é fundamental a utilização de padrões abertos, ou seja, aqueles possíveis de serem implementados em várias linguagens de programação sem dependência de um elemento específico de cada uma.

Quando se trata de sistemas web, ou que precisam ser expostos através da Web para garantir o acesso por clientes geograficamente distantes e, ainda mais, por meio de diferentes dispositivos (smartphones, PCs, tablets, etc.), a utilização de web services tem sido a principal estratégia utilizada há vários anos. Esses serviços atuam como uma camada intermediária entre o banco de dados e os clientes (aplicações que os consomem). Assim, o acesso ao armazenamento de informações, bem como ao processamento lógico inerente ao sistema em questão, é feito por meio desses web services, que oferecem uma interface simplificada e padronizada para consumo por clientes externos. A **Figura 1** ilustra uma visão simplificada desse tipo de arquitetura.

Essa estratégia visa atender à necessidade de interoperabilidade entre sistemas, no entanto, cuidados adicionais precisam ser tomados quando a escalabilidade é um fator crítico. Tomando como base essa figura, é fácil perceber que se por algum motivo o web service ficar indisponível, todo o sistema acaba sendo prejudicado e permanecerá completamente inoperante até que o serviço seja reestabelecido.

Fique por Dentro

Esse artigo apresentará os conceitos da arquitetura REST e como criar um serviço RESTful com DataSnap, demonstrando como utilizar melhor os métodos GET, PUT, POST e DELETE. Esse tema é útil porque permite aos desenvolvedores Delphi construírem, por exemplo, APIs web e sistemas baseados em microsserviços para modularizar grandes projetos e disponibilizá-los na internet. Além disso, os serviços criados seguindo essa arquitetura poderão ser consumidos por aplicações desenvolvidas com diversas linguagens, não se limitando a clientes Delphi.

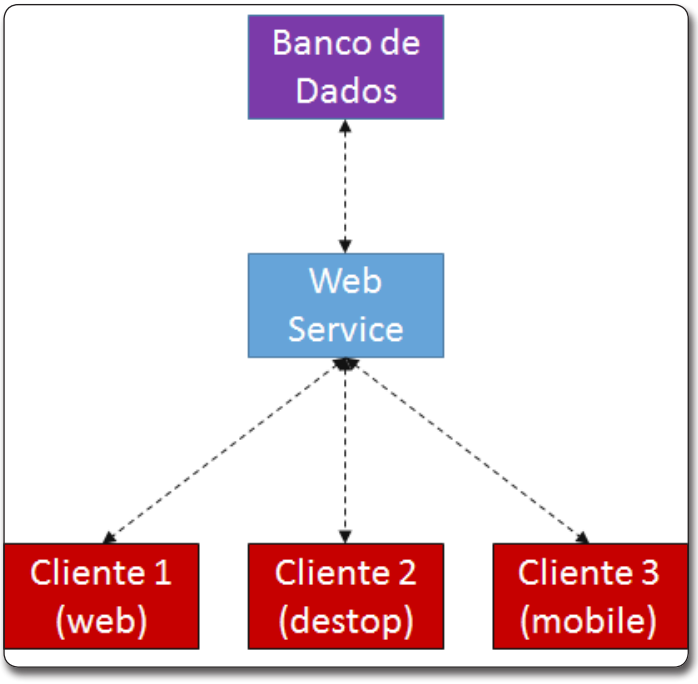


Figura 1. Arquitetura de aplicação utilizando web service

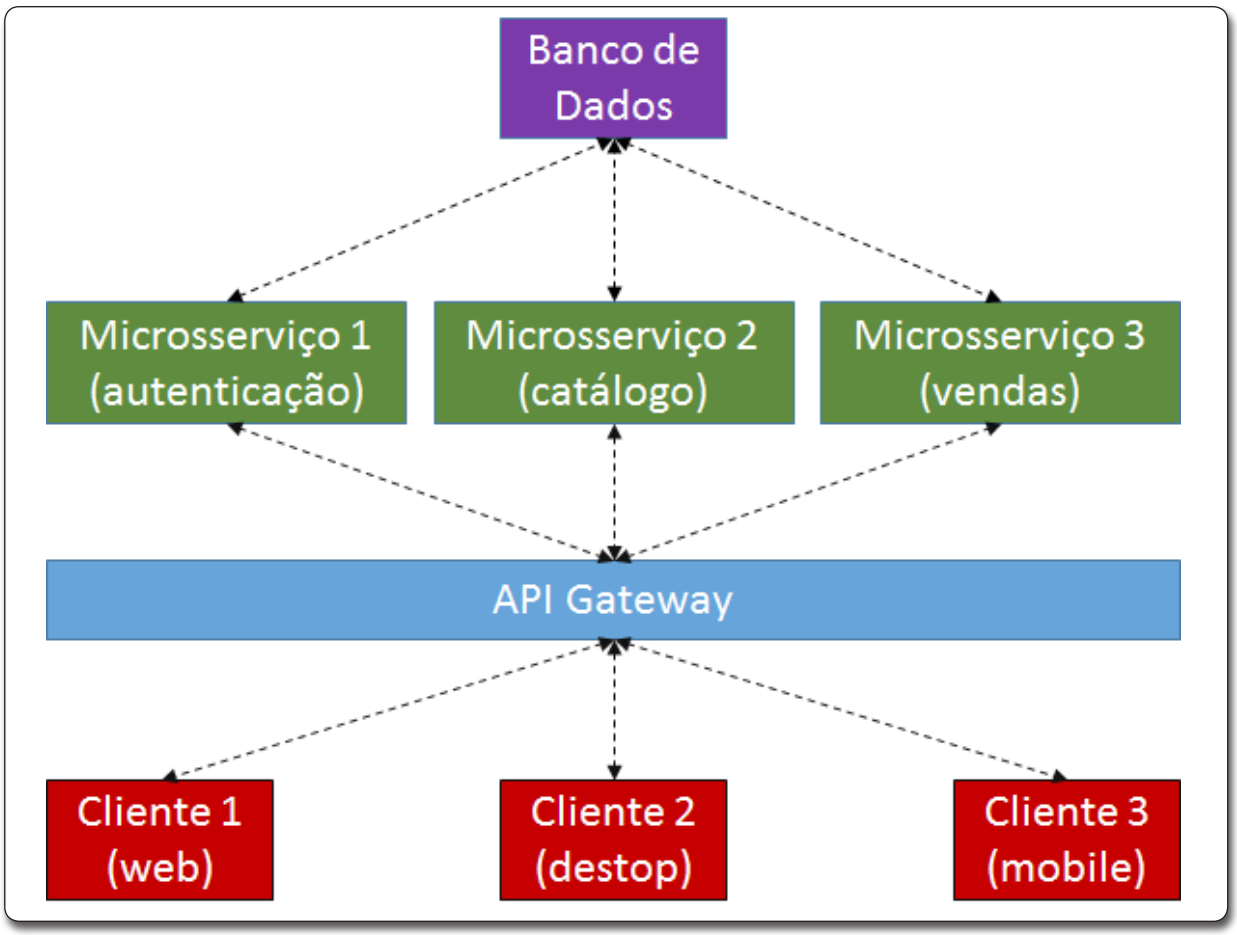


Figura 2. Arquitetura de microsserviços

Nesse contexto surge a arquitetura de microsserviços, que ao invés de manter um único web service para suprir todas as necessidade do sistema, utiliza vários serviços menores, independentes, que proveem funcionalidades específicas. Com isso, a indisponibilidade de um microsserviço não afeta necessariamente os demais. Na **Figura 2** temos uma ilustração desse tipo de arquitetura.

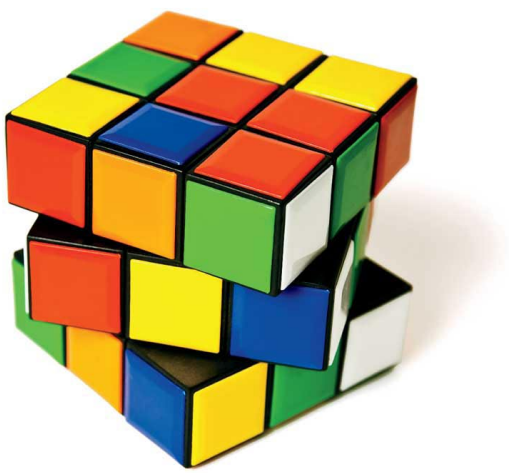
Nessa figura, cada um dos microsserviços é responsável por uma parte específica do sistema (autenticação de usuários, catálogo de produtos e vendas). Se, por algum motivo, o serviço de vendas ficar inoperante, o cliente ainda poderá continuar se autenticando e listando os produtos no site ou aplicativo, por exemplo, pois a infraestrutura utilizada para esse sistema deve permitir e facilitar tal isolamento entre as partes.

Nesse esquema há também a presença da API Gateway, um elemento central que atua orquestrando o acesso aos microsserviços.

Além de garantir flexibilidade e escalabilidade, esse tipo de arquitetura permite que equipes multidisciplinares atuem no mesmo sistema, desenvolvendo os serviços inclusive com linguagens de programação distintas. Aqui retomamos a importância da utilização de padrões abertos, que permitam que apesar de serem feitos em Java, C# ou Delphi, por exemplo, esses serviços possam ser acessados de forma transparente pelos clientes. Nesse contexto, surge o conceito de REST.

REST

REST, sigla de *Representational State Transfer*, é um estilo arquitetural para aplicações web, que utiliza o protocolo HTTP para estabelecer conexão entre diferentes aplicações.



Diferentemente de mecanismos mais complexos como SOAP e WSDL, porém amplamente utilizados, principalmente em aplicações mais antigas, o REST utiliza basicamente os verbos do HTTP para definir as operações que podem ser realizadas pelos clientes. Assim, as operações de CRUD podem ser realizadas através de requisições HTTP utilizando os métodos POST, GET, PUT e DELETE.

Os serviços baseados em REST, chamados de *RESTful services*, devem ser independentes de plataforma e linguagem de programação. Assim, tanto a construção quanto o consumo desse tipo de serviço requerem, basicamente, que a linguagem ofereça suporte à exposição de dados e realização de requisições via protocolo HTTP.

Ao receber uma solicitação do cliente, o servidor deve respondê-la com o recurso solicitado, ou com a informação de que algum erro ocorreu, por exemplo. A resposta, nesse caso, pode ser uma página HTML, uma mensagem em texto simples, um arquivo, entre outros tipos de recursos. Para realizar o tráfego de informações estruturadas, como um objeto do domínio da aplicação (clientes, produtos, etc.), geralmente utiliza-se o formato JSON (*JavaScript Object Notation*). Na **Figura 3** vemos um exemplo de requisição via método GET.

Observe que o cliente fez uma requisição ao servidor por meio de um URI (*Uniform Resource Identifier*), ou seja, um endereço único que identifica o recurso que se está tentando obter. Após receber a solicitação, o servidor possivelmente fez alguma consulta a uma base de dados, obtendo o recurso desejado, e o retornou no formato JSON. Um exemplo comum desse tipo de comunicação ocorre quando acessamos um site através do browser. A resposta, nesse caso, é uma página em HTML, juntamente com alguns recursos adicionais (CSS, JavaScript, imagens, etc.).

Para maiores informações sobre REST, consulte o link disponível no final do artigo, onde há exemplos de implementações em várias linguagens, padrões de design e outras informações pertinentes.

Criando um serviço RESTful com DataSnap

No Delphi, esse tipo de serviço pode ser construído com a tecnologia DataSnap, que vem evoluindo desde a versão 6 do IDE e nas versões mais recentes suporta, além da comunicação via TCP/IP, o desenvolvimento de aplicações baseadas em HTTP e HTTPS. Essa característica torna o DataSnap uma opção bastante viável para programadores Delphi que desejam construir serviços RESTful.

Figura 3. Requisição a serviço RESTful

Criando o projeto

Para explorar essa possibilidade, criaremos nesse artigo um serviço que será responsável por um módulo de cadastro de clientes. Seguindo o que a arquitetura propõe e a demanda geralmente requer, esse serviço compreenderá apenas regras de negócio e armazenamento, sem implementações referentes à camada de apresentação. Toda chamada será feita via protocolo HTTP, seguindo o modelo REST, com troca de informações através de dados nativos ou JSON.

No código fonte, que se encontra disponível para download, há também operações para cadastro de produtos e vendas, complementando o serviço. O Modelo Entidade Relacionamento da base de dados está ilustrado na **Figura 4**.

Para as entidades de Cliente e Produtos o serviço oferecerá opções para adição, atualização, exclusão e listagem de registros. A outra parte compreenderá as entidades Venda e VendaItens.

Utilizaremos um banco de dados SQLite e o utilitário Postman para testar o serviço. Essa ferramenta oferece uma interface simples e intuitiva, além de várias funcionalidades adicionais, para a execução de requisições via HTTP. Na seção **Links** consta o endereço da página oficial do Postman. Instale essa ferramenta para que seja possível efetuar os testes que veremos a seguir nesse artigo.

Para criar o projeto acesse o menu *File* > *New* > *Other*, localize e selecione a opção *DataSnap REST Application* e em seguida clique sobre o botão *OK*. Uma aplicação DataSnap pode ser criada de três formas *Stand-alone VCL Application*, *Stand-alone Console Application* ou *ISAPI Dynamic Link Library*. As opções *Stand-alone* são serviços completamente autossuficientes, seu funcionamento não necessita de servidores como Apache ou ISS. Para serviços criados como *ISAPI* é necessário configurar um servidor.

Para esse exemplo deve ser selecionada a opção *Stand-alone VCL Application*, que por padrão já cria um formulário *Main*. Na próxima tela é solicitada a porta na qual o serviço irá operar e se irá trabalhar sobre o protocolo HTTPS. Nesse caso utilizare-

mos o valor 9999, para evitar conflitos caso a 8080 ou demais comumente utilizadas, como 8081, já esteja em uso. Com a porta definida, clique em *Next*. Na tela seguinte é possível definir algumas características do serviço, como autenticação, métodos de exemplo, arquivos web exemplos, conector mobile e se irá existir um *server module*. Aqui não serão utilizadas essas opções, apenas a *Server Methods Class* antes de pressionar *Next* novamente. Com essa opção marcada, ao criar a aplicação será gerada uma unit para implementação dos serviços.

Na próxima etapa definiremos qual será a classe antecessora daquela que conterá os métodos que serão exportados. Nesse exemplo utilizaremos a TComponent. Por último, deverá ser informado onde serão gravados os arquivos HTML, CSS e JavaScript do projeto. Apesar de eles não serem utilizados nesse projeto, é preciso informar um caminho válido para esse campo, que pode ser o próprio diretório da aplicação. Em seguida, clique em *Finish*.

Após executar todos esses passos, o projeto deve ter uma estrutura semelhante à da **Figura 5**. Para manter o projeto mais limpo, exclua os arquivos da pasta *js*.

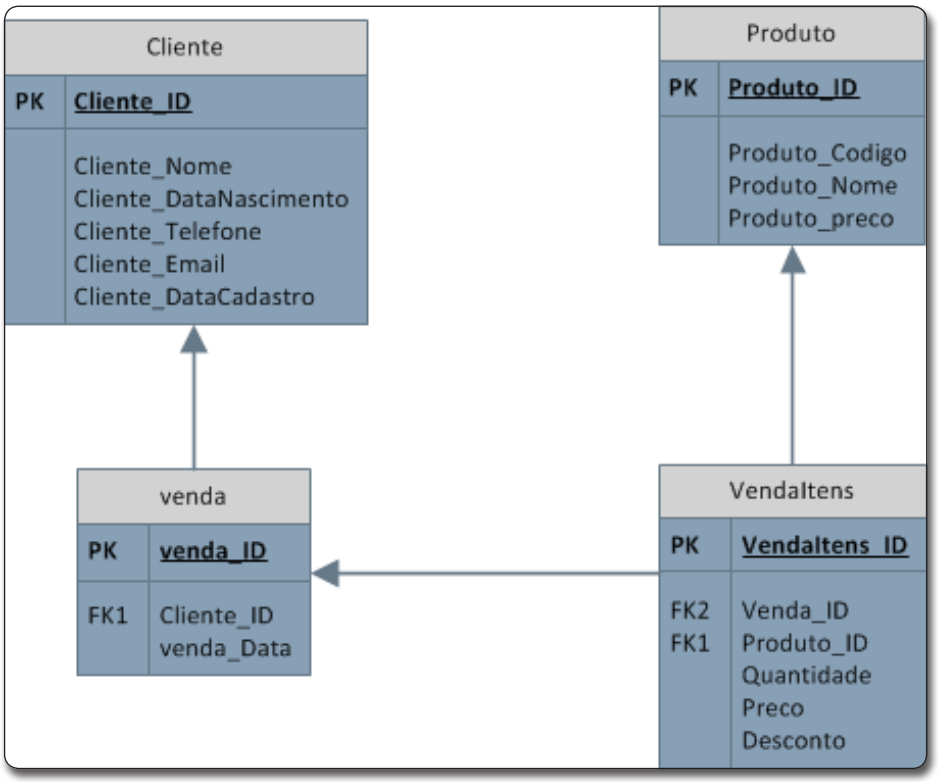


Figura 4. Modelo Entidade Relacionamento do banco de dados

Nota

Serviços em ISAPI podem ser executados em diferentes sistemas operacionais, já serviços Stand-alone só poderão ser executados em ambientes Windows.

Se executarmos o projeto agora já será possível iniciar o servidor da aplicação, simplesmente utilizando a interface do *Form1* que solicita a porta e apresenta botões de *Start* e *Stop*, ou abrir o serviço no browser, como mostra a **Figura 6**

Clique no botão *Start* e em seguida no *Open Browser*. Será aberta uma aba no navegador com a URL *localhost:9999* e a mensagem *DataSnap Server* será apresentada no browser. Sem nenhuma codificação direta já temos um servidor criado e respondendo requisições, porém, ainda sem nenhuma implementação.

Até esse ponto a aplicação deve conter três arquivos .pas:

• *WebModuleUnit1*: serve de repositório para os componentes e métodos que transformam a aplicação em um servidor REST. O wizard insere automaticamente várias estruturas que serão responsáveis por receber, manipular e de responder as requisições. Na **Tabela 1** temos uma breve descrição desses elementos;

• *ServerMethodsUnit1*: serve como repositório para os métodos que serão consumidos. Além disso, essa unit também será utilizada para definição das rotas;

• *FormUnit1:* é responsável por prover a interface principal da aplicação. Nele existem alguns controles que são adicionados pelo wizard e alguns métodos implementados.

Dando continuidade, salve a unit *FormUnit1* como *uDashBoard* e renomeie o formulário para DashBoard.

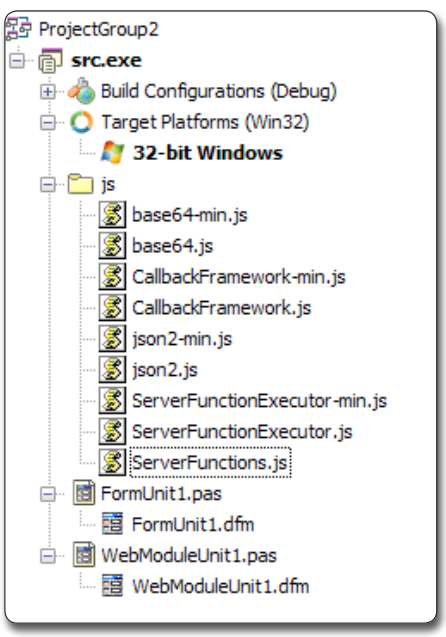


Figura 5. Estrutura do projeto

Componente	Descrição
DSServer1	Responsável por transformar uma aplicação em um servidor.
DSServerClass1	Responsável por "expor" uma classe para que a mesma possa ter seus métodos consumidos pela aplicação cliente. Somente classes que descendem de TPersistent e possuam as diretivas {$MethodInfo ON} e {$MethodInfo OFF} em sua estrutura poderão ser consumidas.
DSHTTPWebDispatcher1	Responsável por receber as requisições feitas ao servidor DataSnap.
DSServerMetaDataProvider1	Responsável pelo intermédio entre um DSServer e um ProxyGeneration, que provê um MetaDataProvider para geração dos proxies.
DSProxyGenerator1	Responsável por gerar as classes proxies para se conectar e consumir um servidor de aplicação.
WebFileDispatcher1	Responsável por gerenciar a requisição aos arquivos e diretórios que compõem o servidor.

Tabela 1. Componentes do WebModules

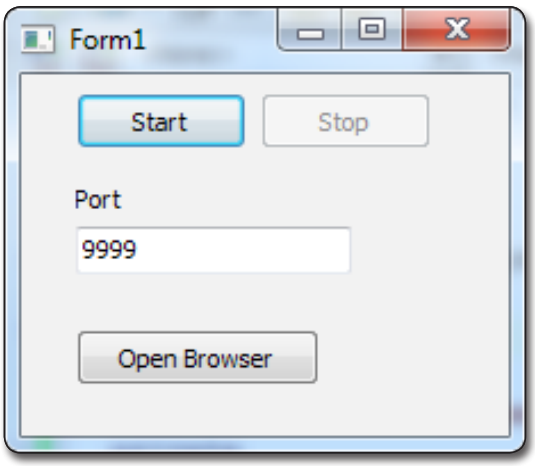


Figura 6. Tela do servidor

Depois, salve a unit ServerMethodsUnit1 como *uService* e renomeie sua classe para *Service*. Por fim, salve o *WebModuleUnit1* como *uWebModule* e renomeie para WebMod.

Como o nome do *ServerMethods* foi alterado e já existia referências a ele em no web module, essas devem ser atualizadas. Altere na seção *uses* o termo *ServerMethodsUnit1* para *uService* e em *DSServerClass1GetClass* altere a referência a *ServerMethodsUnit1.TServerMethods1* para *uService.Service*. Ainda em *uWebModule*, remova os métodos *WebFileDispatcher1BeforeDispatch* e o *WebModule1DefaultHandlerAction*, e os componentes *WebFileDispatcher1*, *DSProxyGenerator1* e o *DSServerMetaDataProvider1*, pois para nossa implementação esses elementos não serão necessários. Assim a aplicação ficará enxuta e mais simples, não acessando nada que não será utilizado.

Se executarmos o servidor agora e abrirmos novamente o navegador, veremos que nada agora é apresentado, pois foi removido o método *DefaultHandler*, que era o responsável por apresentar a mensagem vista anteriormente. Porém, como nenhuma mensagem de erro foi exibida, sabemos que o serviço está respondendo corretamente. Para ter certeza disso, abra o Postman e envie uma requisição GET para o URL *http://localhost:9999*. Para isso, digite esse endereço no campo indicado e clique em *Send*, como pode ser visto na **Figura 7.** A resposta do servidor foi um código 200, indicando que a operação foi bem-sucedida.

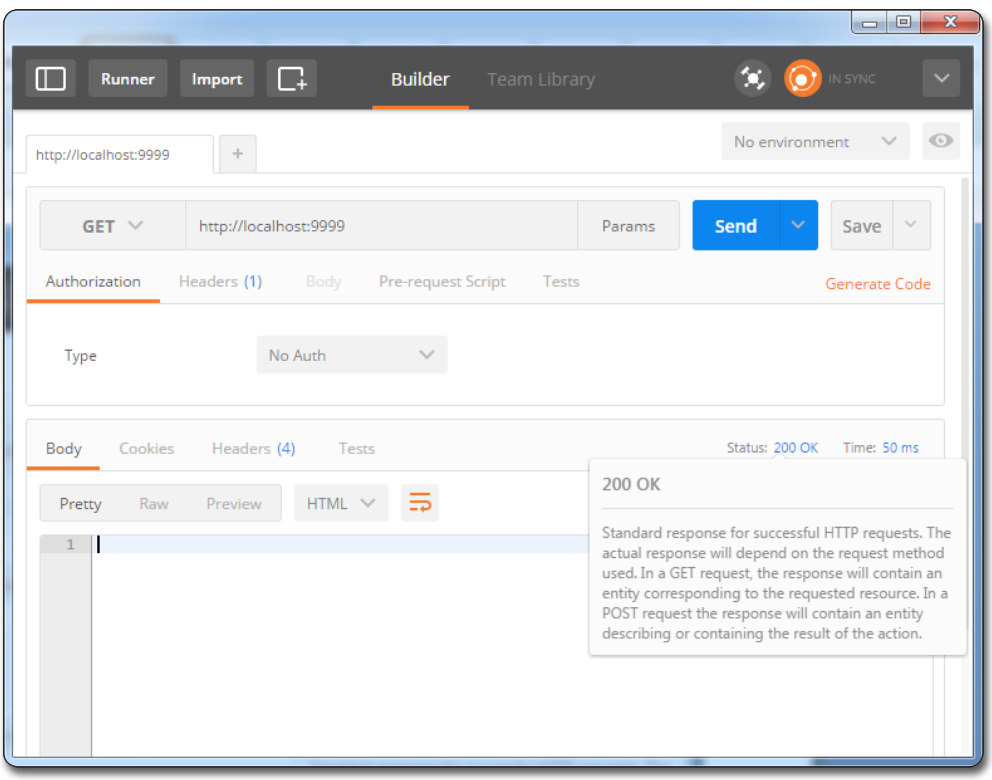


Figura 7. Requisição com status 200 OK no Postman

Além do código 200, o HTTP define vários outros para cada situação como 404 para recurso não encontrado e 500 para erro interno do servidor. Consulte a seção **Links** para os demais códigos de status existentes.

Outro detalhe importante é o URI de acesso aos serviços. Por padrão o Delphi determina que as requisições devem ser feitas através do seguinte

esquema de endereço: *http://Servidor:porta/datasnap/rest/ClassName/MethodName[/inputParameter]**, onde o */datasnap/rest/* é obrigatório, porém pode ser customizado através das propriedades *DSContext* e *RestContext* do componente *TDSHTTPWebDispatcher* que se encontra no *WebModule*, como vemos na **Figura 8.**

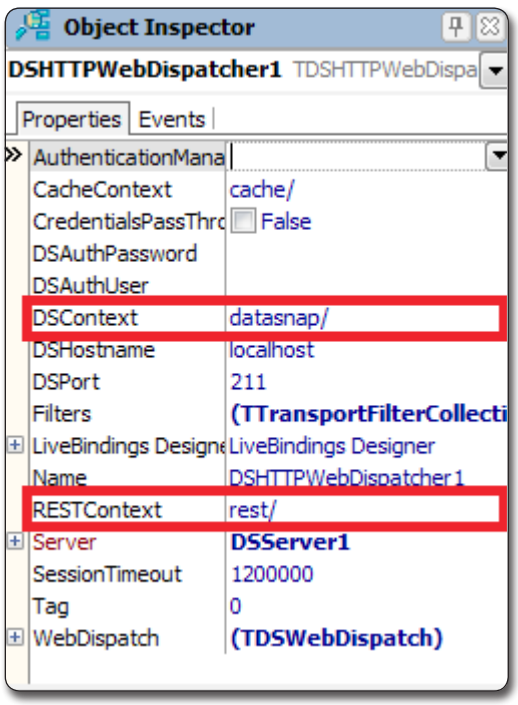


Figura 8. Propriedades de contexto do serviço

Para verificar se o servidor está respondendo corretamente às requisições, adicione em *uService* uma função chamada *Ping* retornando uma *String*. Na implementação desse método adicione a seguinte linha:

```
Result :'OK';
```

Em seguida, excute a aplicação, inicie o serviço e no PostMan faça um GET no URI *http://localhost:9999/datasnap/rest/Service/Ping/*. A seguinte resposta deve ser obtida:

```
{"result":["OK"]}
```

Se esse resultado foi obtido, seu serviço está funcionando corretamente e você já pode avançar para a próxima etapa, onde configuraremos a criação de base de dados pela aplicação e posteriormente a conexão com esse banco para persistir as informações.

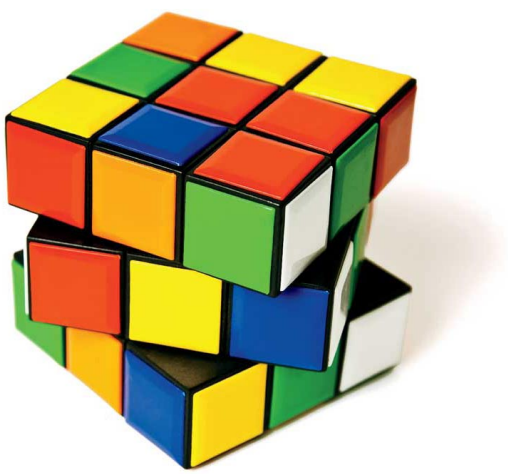
Configurando o acesso ao banco de dados

Crie no projeto uma nova unit e salve como *uMetadados*. Ela será responsável por representar a estrutura do banco de dados. Implemente seu código de acordo com a **Listagem 1**.

Nesse código foram criados quatro *resourcestrings*, cada um apresentando a estrutura de uma tabela, a partir da qual será gerado o objeto equivalente na base de dados.

Listagem 1. Unit de metadata

```
01 unit uMetadados;
02
03 interface
04
05 // Metadados
06 resourcestring
07  // Tabela de Cliente ----------------------------------------------------
08  tbl_Cliente  = 'Cliente (Cliente_ID          INT(10)   ,' +
09                  'Cliente_Nome        VARCHAR(60), ' +
10                  'Cliente_DataNascimento DATE     ,' +
11                  'Cliente_Telefone     VARCHAR(12),' +
12                  'Cliente_Email       VARCHAR(60),' +
13                  'Cliente_DataCadastro  DATE      ' +
14                  ')                          ';
15
16  // Tabela de Produto ----------------------------------------------------
17  tbl_Produtos = 'Produto (Produto_ID     INT      ,' +
18                  'Produto_Codigo  VARCHAR(12),' +
19                  'Produto_Nome   VARCHAR(60),' +
20                  'Produto_preco  REAL      ' +
21                  ')                    ';
22
23
24  // Tabela de Venda ------------------------------------------------------
25  tbl_Venda    = 'Venda (Venda_ID  INT ,' +
26                 'Cliente_ID INT ,' +
27                 'Venda_Data DATE ' +
28                 ')          ';
29
30  // Tabela de VendaItens -------------------------------------------------
31  tbl_VendaItens = 'VendaItens (VendaItens_ID  INT  ,' +
32                    'Venda_ID      INT  ,' +
33                    'Produto_ID    INT  ,' +
34                    'Quantidade    INT  ,' +
35                    'Preco         FLOAT,' +
36                    'Desconto      FLOAT ' +
37                    ')             ';
38
39
40  implementation
41
42 end.
```



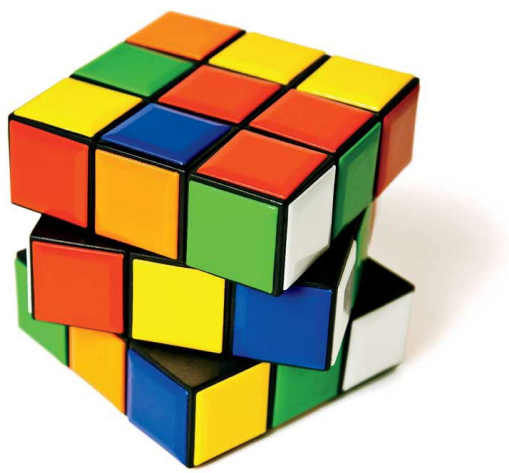
Adicione em seguida mais uma unit e a nomeie como *uUtilDB*. Essa será responsável por configurar a conexão com banco de dados, criar as tabelas e recuperar um Id para uma tabela. Nessa unit criaremos uma classe chamada *TConexaoDB*, cuja estrutura é apresentada na **Listagem 2**.

Listagem 2. Estrutura da classe TConexaoDB

```
01  TConexaoDB = class
02    private
03      fConnection : TFDConnection;
04    protected
05      procedure ConfigurarDB;
06      procedure ConfigurarTabelas;
07      procedure CriarTabela(sTabela : String);
08    public
09      constructor Create;
10      destructor Destroy;
11
12      function NovoID(psTabela, psId : String) : Integer;
13    published
14      property Connection : TFDConnection read fConnection;
15  end;
```

A seguir, veremos em detalhes cada um dos métodos dessa classe e suas responsabilidades, a começar pelo *ConfigurarDB*, responsável por passar os parâmetros para a conexão. Esse método é executado no construtor da classe, logo após se instanciar o *fConnection*. Veja na **Listagem 3** a sua implementação.

Na linha 9 definimos o caminho do banco de dados, que vai ficar dentro de uma pasta chamada *db* junto à aplicação. Na linha 11 verificamos se o arquivo já existe e em caso negativo, ele é criado. Em seguida a flag *configuraTabelas* é definida como *true*. Da linha 19 a 23 são passados os parâmetros para a conexão e na linha 24 a conexão é estabelecida. Já na linha 27 verificamos se a flag *configuraTabelas* está como true e em caso afirmativo chamamos o método *configurarTabelas*, que é responsável por definir quais tabelas serão criadas no banco e sua implementação pode ser vista na **Listagem 4**.



Listagem 3. Método de configuração de conexão

```
01 procedure TConexaoDB.ConfigurarDB;
02 Const
03   sNameDB = 'db\DB_AppREST.s3db';
04 Var
05   sDbPath       : String;
06   configuraTabelas : Boolean;
07   handleFile     : Integer;
08 begin
09   sDbPath := ExtractFilePath(Application.ExeName) + sNameDB ;
10
11   if Not(FileExists(sDbPath)) then
12   Begin
13     handleFile := FileCreate(sDbPath);
14     FileClose(handleFile);
15     configuraTabelas := True;
16   End;
17
18   // Passa os parametros de Conexão e Conecta ao Banco de Dados
19   Connection.LoginPrompt := False;
20   Connection.Params.Clear;
21   Connection.Params.Values['Database']    := sDbPath;
22   Connection.Params.Values['DriverID']    := 'SQLite';
23   Connection.Params.Values['CharacterSet'] := 'utf8';
24   Connection.Connected := True;
25
26   // Verifica Tabelas
27   if configuraTabelas then
28     ConfigurarTabelas;
29 end;
```

Listagem 4. Método que define as tabelas a serem criadas

```
01 procedure TConexaoDB.ConfigurarTabelas;
02 begin
03   CriarTabela(tbl_Cliente   );
04   CriarTabela(tbl_Produtos  );
05   CriarTabela(tbl_Venda     );
06   CriarTabela(tbl_VendaItens);
07 end;
```

A implementação desse método é bem simples: ele apenas faz chamada ao método *CriarTabela*, passando como parâmetro a constante da estrutura desejada que se encontra na unit *uMetadados*. O método *CriarTabela* é responsável por executar os comandos DDL (*Data Definition* Language) no banco de dados, como mostra a **Listagem 5**.

Nesse método é instanciado um objeto do tipo *TFDQuery*, que faz executa o comando *CREATE TABLE* a partir das definições passadas no parâmetro *sTable*. Como o banco que está sendo utilizado não dispõe de recursos de para geração automática de valores, precisaremos implementar um novo método chamado NovoID, de acordo com a **Listagem 6**.

Esse método obtém como resultando o valor máximo da consulta feita na tabela mais 1, gerando assim um novo ID para a tabela e coluna passada nos parâmetros *psTabela* e *psId*. Com ele, a classe está finalizada, então adicione uma variável global com nome

de *ConexaoDB* do tipo *TConexaoDB* e no *OnCreate* do *Dashboard* adicione a seguinte linha:

```
ConexaoDB := TConexaoDB.Create;
```

Listagem 5. Método para criar as tabelas

```
01 procedure TConexaoDB.CriarTabela(sTabela: String);
02 Var QueryValid : TFDQuery;
03 begin
04   QueryValid := TFDQuery.Create(Connection);
05   try
06     QueryValid.Connection := Connection;
07     QueryValid.SQL.Clear;
08     QueryValid.SQL.Text := 'CREATE TABLE ' + sTabela;
09     QueryValid.ExecSQL;
10   finally
11     FreeAndNil(QueryValid);
12   end;
13 end;
```

Listagem 6. Método para Gerar Ids sequenciais

```
01 function TConexaoDB.NovoID(psTabela, psId: String): Integer;
02 Var
03   oQuery : TFDQuery;
04 begin
05   oQuery := TFDQuery.Create(Connection);
06   try
07     oQuery.Connection := Connection;
08
09     oQuery.SQL.Clear;
10     oQuery.SQL.Text := 'Select coalesce(Max('+psId+'),0) id From ' + psTabela;
11     oQuery.Open;
12
13     Result := oQuery.FieldByName('id').AsInteger + 1;
14   finally
15     FreeAndNil(oQuery);
16   end;
17 end;
```

Para se trabalhar com componentes de acesso ao banco da biblioteca FireDac, deve existir um componente *TFDGUIxWaitCursor* na aplicação. Adicione-o então ao form *Dashboard* e execute a aplicação. Você perceberá que um arquivo *DB_AppREST.s3db* foi criado na pasta *db*.

Com a conexão com o banco de dados estabelecida, podemos agora programar a persistência de dados e regras de negócio referentes ao cadastro de cliente, que é a funcionalidade principal do nosso serviço.

Criando o cadastro de clientes

Crie uma nova unit chamada *uCliente*, que será responsável por manipular os registros do cliente. Crie nela uma classe chamada de *TCliente*, seu cuja assinatura deve corresponder ao que mostra a **Listagem 7**.

Essa classe é responsável pelas consultas e persistência ao banco de dados. Nela foram adicionados os elementos da tabela de modo público, sem nenhum encapsulamento, para simplificar o código

Os métodos de retornar e deletar registros são estáticos, assim, não é preciso instanciar um objeto apenas para consultar ou excluir dados. Entretanto, para salvar um elemento deve-se criar uma instância, visto que esse método irá refletir os dados do próprio objeto no banco de dados. Para facilitar a implementação, existe publicamente apenas o método *Salvar*, que irá verificar se o Id do cliente está preenchido. Se sim, é considerado como um *update* e chamando o método *Atualizar* que está privado, caso contrário é chamando o método *Adicionar* que também se encontra na seção *private*. O código desse método pode ser visto na **Listagem 8.**

Listagem 7. Prototipo classe TCliente

```
01  TCliente = class
02    private
03      function Adicionar : Boolean;
04      function Atualizar : Boolean;
05    public
06      Cliente_ID          : Integer;
07      Cliente_Nome        : String ;
08      Cliente_DataNascimento : TDate  ;
09      Cliente_Telefone    : String ;
10      Cliente_Email       : String ;
11      Cliente_DataCadastro  : TDate  ;
12
13      class function RetornarTodosRegistros : tDataSet;
14      class function RetornarRegistroById(pnId : Integer) : TDataSet;
15      class procedure DeletarRegistroById(pnId : Integer);
16
17      function Salvar : Boolean;
18
19      procedure Assigned(json : TJSONObject);
20  end;
```

Listagem 8. Método para salvar cliente

```
01 function TCliente.Salvar: Boolean;
02 begin
03   if Cliente_ID > 0 then
04     Atualizar
05   else
06     Adicionar;
07 end;
```

Esse método é bem simples, visto que atua apenas como uma fachada para a chamada aos demais. As operações concretas de acesso ao banco de dados ficarão nos métodos *Atualizar* e *Adicionar*, que pode ser visto na **Listagem 9**.

Apesar de um pouco extenso, esse método é muito simples: primeiramente instanciamos um objeto do tipo *TFDQuery*, que é ligado à conexão que está definida na instância de *ConexaoDB*. Em seguida, da linha 10 a 24, é passado o comando SQL de *INSERT* e os parâmetros. Da linha 26 a 31 são atribuídos os valores do objeto cliente aos parâmetros da query e na linha 32 é executado o comando SQL no banco de dados.

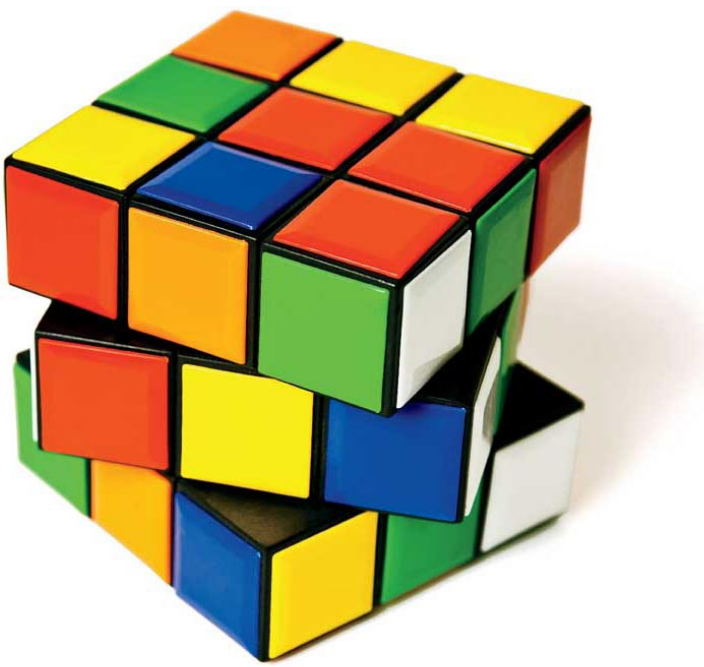
Como o objeto cliente será recebido em formato JSON, será criada uma classe para passar os valores do JSON para o objeto. Para isso, implemente o método *Assign* de acordo com a **Listagem 10**.

Nesse método apenas vinculamos aos campos do objeto os seus respectivos valores no JSON. Com isso, já estamos próximos de conseguir criar um registro no banco através do serviço, mas antes é necessário exportar o método de adição de clientes. Para isso, vá à unit *uService*, e implemente o método de adicionar que será chamando através da URI. Antes disso, no entanto, é preciso compreender como o Delphi mapeia os métodos para cada operação. Inicialmente criamos o método *Ping* e o invocamos através de um GET, porém, para as demais operações isso não é possível. Para cada uma das delas um prefixo deve ser adicionado para que seja feito o mapeamento adequadamente à sua ação, conforme consta na **Tabela 2**.

Operação	Prefixo	Exemplo
GET		Recurso
PUT	accept	acceptRecurso
POST	update	updateRecurso
DELETE	cancel	cancelRecurso

Tabela 2. Prefixos de operações

Apenas a chamada de GET é feita diretamente ao nome real da função, as demais operações são automaticamente mapeadas para o método devidamente prefixado. Caso se tente fazer uma chamada de uma operação que não esteja mapeada, será retornado um erro informando que o método não foi encontrado na lista de serviços.



Listagem 9. Método para adicionar cliente

```
01 function TCliente.Adicionar: Boolean;
02 Var
03   oQuery : TFDQuery;
04 begin
05   oQuery := TFDQuery.Create(ConexaoDB.Connection);
06   try
07    try
08     oQuery.Connection := ConexaoDB.Connection;
09     oQuery.SQL.Clear;
10     oQuery.SQL.Add('Insert into Cliente (     ');
11     oQuery.SQL.Add('   Cliente_ID         , ');
12     oQuery.SQL.Add('   Cliente_Nome       , ');
13     oQuery.SQL.Add('   Cliente_DataNascimento, ');
14     oQuery.SQL.Add('   Cliente_Telefone   , ');
15     oQuery.SQL.Add('   Cliente_Email      , ');
16     oQuery.SQL.Add('   Cliente_DataCadastro  ');
17     oQuery.SQL.Add(')values(                 ');
18     oQuery.SQL.Add('   :Cliente_ID        ,');
19     oQuery.SQL.Add('   :Cliente_Nome      ,');
20     oQuery.SQL.Add('   :Cliente_DataNascimento,');
21     oQuery.SQL.Add('   :Cliente_Telefone  ,');
22     oQuery.SQL.Add('   :Cliente_Email     ,');
23     oQuery.SQL.Add('   :Cliente_DataCadastro  ');
24     oQuery.SQL.Add(')                        ');
25
26     oQuery.ParamByName('Cliente_ID').AsInteger := ConexaoDB.
       NovoID('Cliente','Cliente_ID');
27     oQuery.ParamByName('Cliente_Nome').AsString  := Cliente_Nome;
28     oQuery.ParamByName('Cliente_DataNascimento')
       .AsDate := Cliente_DataNascimento;
29     oQuery.ParamByName('Cliente_Telefone').AsString  := Cliente_Telefone;
30     oQuery.ParamByName('Cliente_Email').AsString  := Cliente_Email;
31     oQuery.ParamByName('Cliente_DataCadastro' ).AsDateTime:= Now();
32     oQuery.ExecSQL;
33
34     Result := true;
35    except
36     Result := false;
37    end;
38   finally
39    FreeAndNil(oQuery);
40   end;
41 end;
```

Listagem 10. Método Assign

```
01 procedure TCliente.Assign(json: TJSONObject);
02 begin
03   Cliente_ID  := json.GetValue('Cliente_ID').Value.ToInteger;
04   Cliente_Nome := json.GetValue('Cliente_Nome').Value;
05   Cliente_DataNascimento := StrToDate(json.GetValue(
       'Cliente_DataNascimento').Value);
06   Cliente_Telefone := json.GetValue('Cliente_Telefone').Value;
07   Cliente_Email   := json.GetValue('Cliente_Email').Value;
08 end;
```

Para adicionar um recurso, o verbo do HTTP utilizado nesse caso será o PUT e ao criar esse método devemos nomeá-lo como *accept* + o nome do recurso desejado. Assim sendo, crie o método *acceptCliente* e o implemente de acordo com a **Listagem 11**.

Essa função é bastante simples, visto ela apenas faz uma chamada ao *ClienteSalvar*, que deve ser criado como *private* na classe *Service*. Esse método foi criado com o intuito de evitar a redundância de código, uma vez que será o funcionamento da atualização será a mesma. Na **Listagem 12** vemos como fica sua implementação.

Listagem 11. Método acceptCliente

```
01 function Service.acceptCliente(jsonCliente : TJSONObject): String;
02 Begin
03   ClienteSalvar(jsonCliente);
04 end;
```

Listagem 12. Método ClienteSalvar

```
01 function Service.ClienteSalvar(jsonCliente : TJSONObject): Boolean;
02 Var Cliente : TCliente;
03 begin
04   Cliente := TCliente.Create();
05   try
06     Cliente.Assign (jsonCliente);
07     Cliente.Salvar;
08   finally
09     Cliente.Free;
10   end;
11 end;
```

Nesse método criamos uma instância da classe *TCliente* e usamos a função *Assign* para transferir os valores do JSON recebido o objeto criado. Em seguida, efetuamos a operação de salvar o cliente criado.

Para testar o serviço, basta executar a aplicação, iniciar o serviço e no PostMan enviar uma requisição para o seguinte URL: *http://localhost:9999/datasnap/rest/Service/Cliente/*. Ao lado esquerdo da barra de endereço existe um dropdown para escolher as operações. Selecione PUT, vá na aba *Body* e marque o radiobutton *raw*. Em seguida, onde está selecionado *Text*, escolha a opção *JSON(application/json)* e no campo abaixo adicione o conteúdo em formato JSON que é apresentado na **Listagem 13**. Sua tela deve estar semelhante à **Figura 9**.

Clique no botão *Send* e perceba que em *Status* será apresentado o código 201 com a mensagem *Created*. Isso indica que a operação foi executada com sucesso. Adicione mais alguns registros para que seja possível realizar algumas consultas posteriormente.

Feito isso, vá até a classe *TCliente* e implemente o método *RetornarTodosRegistros*, conforme mostra a **Listagem 14**.

Nessa função fazemos um *SELECT* no banco, retornando todos os registros da tabela *Cliente* através do objeto do tipo *TFDQuery*.

Seguindo essa mesma lógica, implementaremos agora o método *RetornarRegistroById*, cujo código encontra-se na **Listagem 15**.

Listagem 13. Dados do cliente em formato JSON

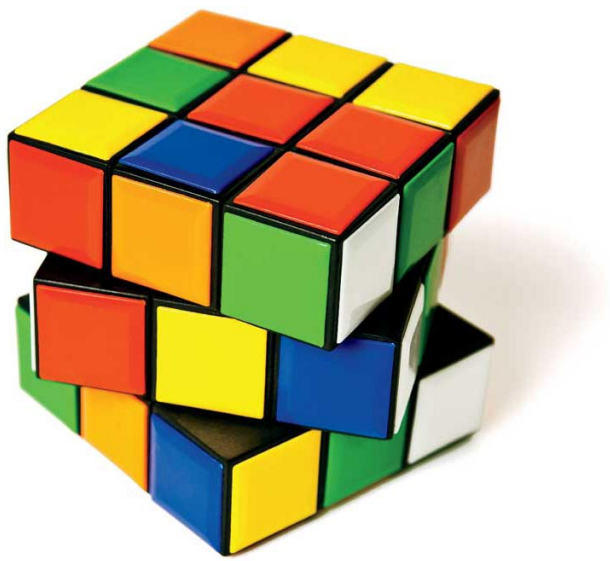
```
01 {
02   "Cliente_ID":0,
03   "Cliente_Nome":"Meu Novo Cliente",
04   "Cliente_DataNascimento":"05/08/1989",
05   "Cliente_Telefone":"9999-9999",
06   "Cliente_Email":"NovoCliente@hotmail.com"
07 }
```

Listagem 14. Método para retornar todos os clientes

```
01 class function TCliente.RetornarTodosRegistros: tDataSet;
02 Var
03   oQuery : TFDQuery;
04 begin
05   oQuery := TFDQuery.Create(ConexaoDB.Connection);
06
07   oQuery.Connection := ConexaoDB.Connection;
08   oQuery.SQL.Clear;
09   oQuery.SQL.Text := 'Select * From Cliente';
10   oQuery.Open;
11
12   Result := oQuery;
13 end;
```

Listagem 15. Método para retornar um cliente por Id

```
01 class function TCliente.RetornarRegistroById(pnId : Integer): TDataSet;
02 Var
03   oQuery : TFDQuery;
04 begin
05   oQuery := TFDQuery.Create(ConexaoDB.Connection);
06
07   oQuery.Connection := ConexaoDB.Connection;
08   oQuery.SQL.Clear;
09   oQuery.SQL.Add('Select * From Cliente      ');
10   oQuery.SQL.Add('Where Cliente_ID = :Cliente_ID');
11
12   oQuery.ParamByName('Cliente_ID').AsInteger :=  pnId;
13   oQuery.Open;
14
15   Result := oQuery;
16 end;
```



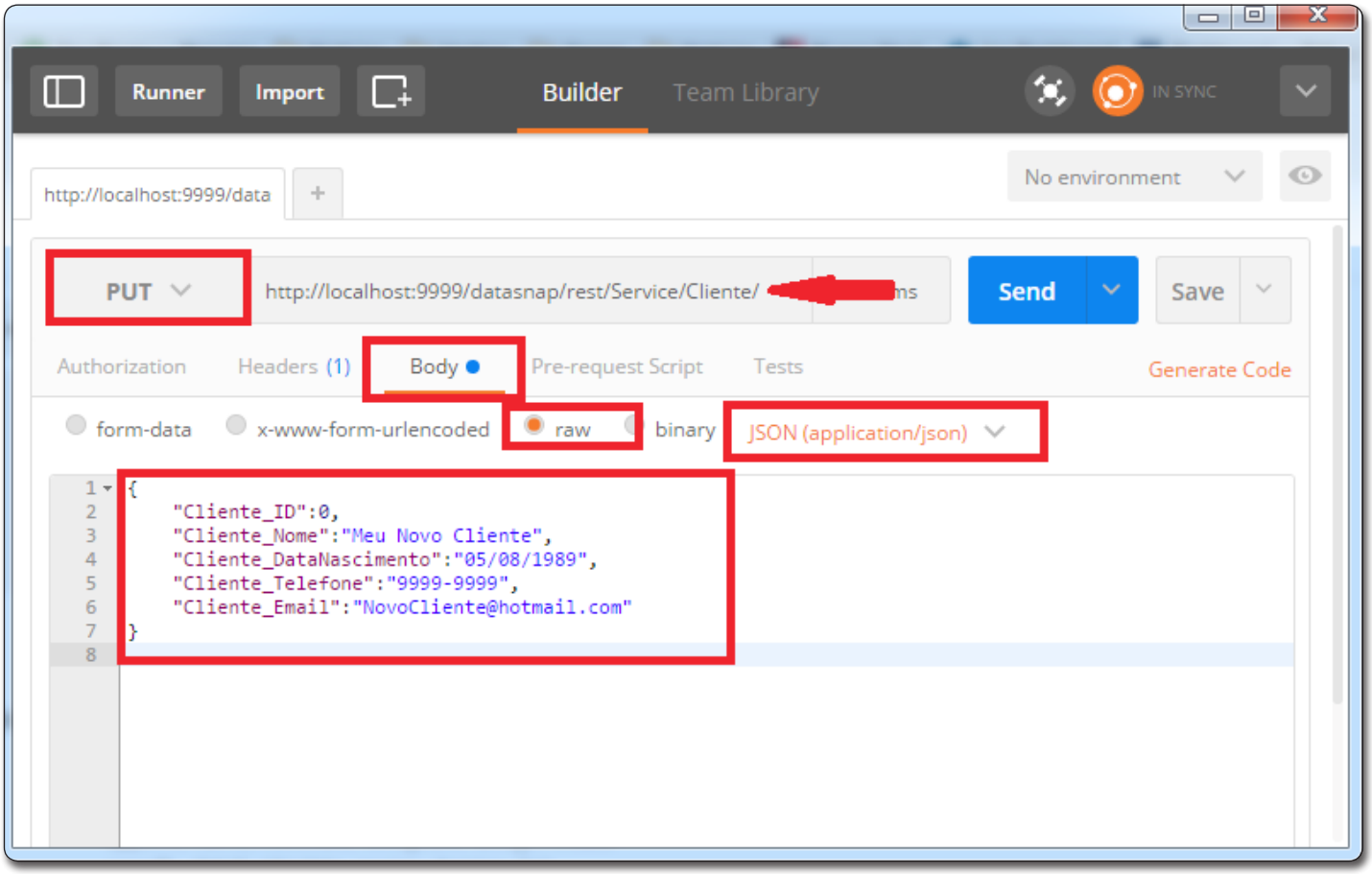
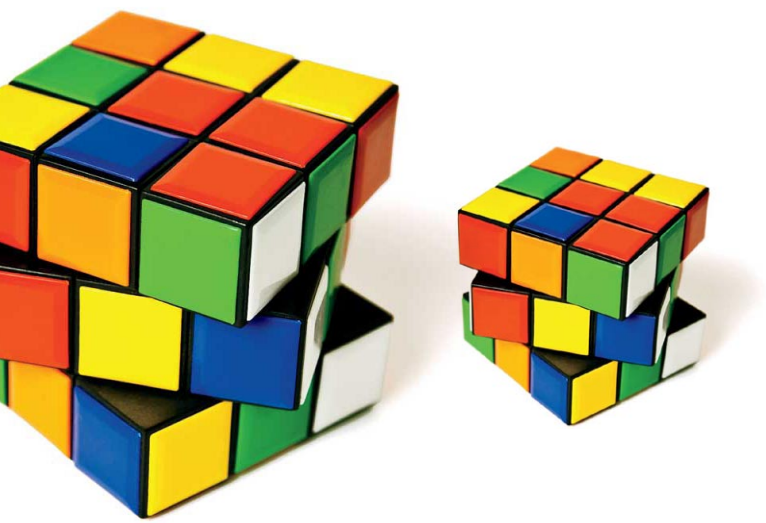


Figura 9. Enviando requisição PUT com PostMan

A única diferença nesse caso é a presença da cláusula *WHERE* na consulta, a fim de filtrar um único cliente.

Voltemos agora para a unit *uService*, na qual precisamos disponibilizar o serviço de consulta. Como os métodos que acabamos de implementar retornam objetos do tipo *TFDQuery* e para o serviço REST o resultado deve estar em formato JSON, precisaremos criar uma função intermediária para efetuar essa conversão. Para isso, temos na **Listagem 16** o método *DataSetToJSON*.



Listagem 16. Método para serializar um dataset em formato JSON

```
01 function DataSetToJSON(oDataSet : TDataset) : TJSONArray;
02 Var
03   JObject  : TJSONObject;
04   i        : Integer;
05 begin
06   Result   := TJSONArray.Create;
07   oDataSet.First;
08   while Not(oDataSet.Eof) do
09   begin
10     JObject := TJSONObject.Create;
11     for i := 0 to oDataSet.FieldCount-1 do
12       JObject.AddPair(oDataSet.Fields[i].FieldName,
         TJSONString.Create(oDataSet.Fields[i].AsString));
13
14     Result.AddElement(JObject);
15     oDataSet.Next;
16   end;
17 end;
```

Na linha 6 criamos um objeto do tipo *TJSONArray*, que conterá uma coleção de objetos do tipo *TJSONObject* e será retornado pelo método. Entre as linhas 8 e 16 percorremos o dataset, criando um novo elemento, cujas propriedades serão preenchidas com as colunas da tabela.

Com isso já podemos criar, na classe *Service*, o método que responderá às requisições através do verbo GET e leva apenas

o nome do recurso a ser buscado. Na **Listagem 17** temos essa implementação.

Na linha 3 verificamos se o argumento recebido é maior que zero. Em caso positivo, retornarmos apenas o cliente filtrado pelo Id (linha 4). Caso contrário, retornarmos todos os clientes (linha 6).

Listagem 17. Método de consulta de clientes

```
01 function Service.Cliente(pnIdCliente : Integer): TJSONArray;
02 begin
03   if pnIdCliente > 0 then
04     result := DataSetToJSON(TCliente.RetornarRegistroById(pnIdCliente))
05   else
06     result := DataSetToJSON(TCliente.RetornarTodosRegistros);
07 end;
```

Execute a aplicação, inicie o serviço e retorne ao PostMan para consultar os dados de todos os clientes. Envie uma requisição GET para o URL *http://localhost:9999/datasnap/rest/Service/Cliente/* para listar todos e, para filtrar apenas um registro, adicione o Id desejado no final do endereço: *http://localhost:9999/datasnap/rest/Service/Cliente/5/*.

Retornaremos agora ao código da unit *uSevice* para implementar os métodos de atualização e exclusão de clientes. O método *updateCliente*, cujo código encontra-se na **Listagem 18**, funciona da mesma forma que o *acceptCliente*. Para testá-lo, envie agora uma requisição POST no PostMan, passando os dados do cliente no formato JSON, só que agora incluindo o Id do registro já existente. Para verificar se a atualização foi realizada corretamente, faça uma consulta via método GET informando o Id do objeto que acabou de alterar.

Listagem 18. Método updateCliente

```
01 function Service.updateCliente(jsonCliente : TJSONObject): String;
02 begin
03   ClienteSalvar(jsonCliente);
04 end;
```

Para concluir as operações de CRUD do cliente, resta implementar agora a exclusão. Para isso, devemos implementar na classe *TCliente* o procedimento *DeletarRegistroById*, cujo código encontra-se na **Listagem 19**.

Nesse método, executamos uma instrução *DELETE* no banco de dados, passando como parâmetro o Id do cliente que foi recebido no argumento *pnId*. Para expor essa operação para clientes externos, temos de voltar à classe *Service* e implementar o método *cancelCliente* de acordo com a **Listagem 20,** seguindo o padrão de nomenclatura indicado na **Tabela 2.**

Para testar essa operação, devemos selecionar no PostMan o verbo *DELETE*, e indicar no URL o Id do cliente a ser removido da base de dados. Após isso, podemos enviar uma requisição GET sem nenhum Id, para que todos os registros sejam listados.

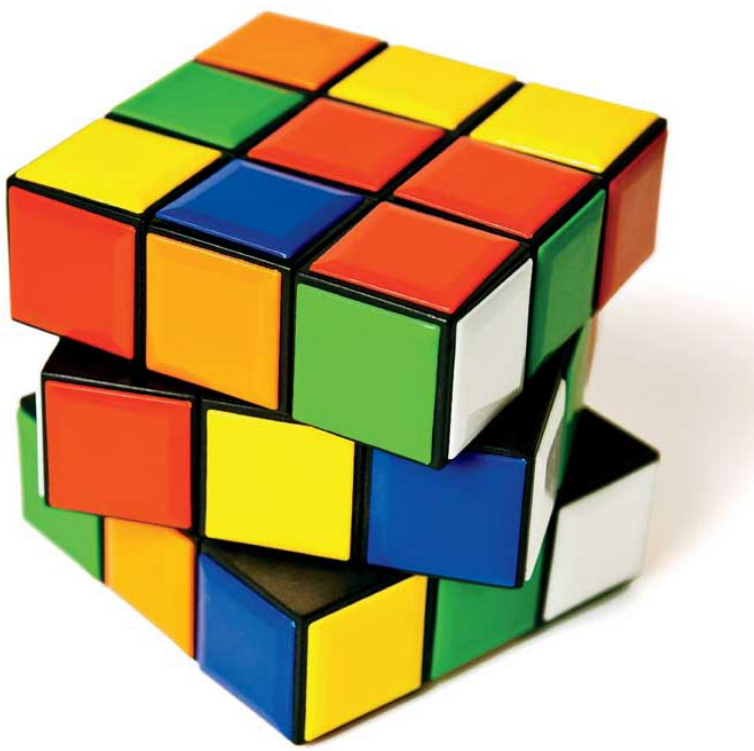
Com isso concluímos o conjunto de operações sobre a entidade *Cliente* que serão expostas para os clientes através do serviço RESTful. Por meio de requisições HTTP qualquer outra aplicação estará apta a se comunicar com nosso sistema, seja ela web, desktop ou mobile, e ainda desenvolvida com qualquer outra linguagem.

Listagem 19. Método de exclusão de clientes

```
01 class procedure TCliente.DeletarRegistroById(pnId: Integer);
02 Var
03   oQuery : TFDQuery;
04 begin
05   oQuery := TFDQuery.Create(ConexaoDB.Connection);
06
07   oQuery.Connection := ConexaoDB.Connection;
08   oQuery.SQL.Clear;
09   oQuery.SQL.Add('Delete From Cliente        ');
10   oQuery.SQL.Add('Where Cliente_ID = :Cliente_ID');
11
12   oQuery.ParamByName('Cliente_ID').AsInteger :=  pnId;
13   oQuery.ExecSQL;
14 end;
```

Listagem 20. Método cancelCliente

```
01 function Service.cancelCliente(pnIdCliente : Integer): String;
02 begin
03   TCliente.DeletarRegistroById(pnIdCliente);
04 end;
```



Para aplicações mais complexas, com várias entidades, é aconselhado separar as responsabilidades das classes, mantendo as operações de acesso ao banco de dados separadas das definições de características e validações, por exemplo.

Também é uma boa prática, nesses casos, oferecer para o cliente uma documentação do seu serviço, de forma que seja possível identificar para qual URL devem ser enviadas as requisições para cada finalidade. Documentações desse tipo geralmente incluem o endereço do recurso, os métodos disponíveis, com seus parâmetros, retornos esperados e erros possíveis.

Ao desenvolver serviços desse tipo conseguimos isolar as regras de negócio do *front-end* da aplicação e aumentar o poder de escalabilidade. Assim, podemos com maior facilidade desenvolver novas versões do sistema para diferentes plataformas. Por exemplo, a aplicação que antes estava disponível apenas para ambiente Windows, poderá ter uma versão para Android criada rapidamente. Caberá ao desenvolvedor focar-se no desenvolvimento da interface gráfica e questões específicas da plataforma, uma vez que as operações de persistência de dados e algumas regras complexas já estarão implementadas no *back-end* e serão expostas por meio de um serviço REST.

Autor

Gutierry Antonio
gutierrydsn@hotmail.com

Entusiasta de Big Data e Data Warehouses, atua como Engenheiro de Software e DBA MySQL. É graduado em Sistemas da Informação e Análise de Sistemas, e atualmente cursa especialização em Engenharia de Sistemas. Possui experiência em desenvolvimento com Delphi e Ruby on Rails com bancos de dados Firebird e MySQL, além de conhecimento de Java, C# e C++ com SQL Server e Oracle, bem como de AWS.

Links:

Postman
https://www.getpostman.com/

Descrição e tutoriais sobre REST
http://rest.elkstein.org/

Códigos de Status do HTTP
http://www.restapitutorial.com/httpstatuscodes.html

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Guia HTML 5

Um verdadeiro manual de referência com tudo que você precisa sobre HTML!

DEVMEDIA

http://www.devmedia.com.br/guias/guia-html/3

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486